DEPARTMENT OF THE AIR FORCE
HQ Air Force Operational Test and Evaluation Center

AFOTEC PAMPHLET 99-102, Volume 3
1 September 1996

Test and Evaluation

SOFTWARE MAINTAINABILITY EVALUATION GUIDE

The purpose of this pamphlet is to provide Air Force Operational Test and Evaluation Center (AFOTEC) personnel information needed to evaluate the maintainability of a software product. This volume describes how to plan, conduct and report a software maintainability evaluation, and contains standardized questionnaires that provide a framework for obtaining software maintenance personnel ratings of the maintainability of source code listings, maintenance documentation, and the implementation (design).

This volume is number three in a series of software operational test and evaluation guides prepared by the Software Analysis Team at Headquarters (HQ) AFOTEC. Local reproduction of all volumes in this series is authorized. This volume is an evolutionary document that will be updated periodically. Comments should be directed to the office of primary responsibility (OPR).

SUMMARY OF CHANGES

The entire document has been re-written.

Supersedes AFOTECP 800-2, Volume 3, 17 Sep 91
OPR: AFOTEC/SAS (Capt Ray Lewis)

Certified by: AFOTEC/SA (Lt Col F. Swehosky)
Pages: 164/~~Distribution F~~

TABLE OF CONTENTS

ATTACHMENTS

FIGURES

TABLES

1. Overview. This pamphlet describes how to plan, conduct, and report a software maintainability evaluation in support of operational test and evaluation. Contained in this publication are standardized questionnaires used to evaluate the maintainability of a software product. Volume 1 of this pamphlet explains how software maintainability evaluations fit into the overall AFOTEC concept of software OT&E, and this overview section provides additional detail. Typically, a software maintainability evaluation involves a number of people filling the roles of Software Test Manager (STM), Deputy for Software Evaluation (DSE), Moderator, and Evaluators. One or more of these roles may be filled by the same person. Volume 1 explains these roles and associated responsibilities within the context of an overall OT&E program. This evaluation guide provides further detail on how these roles apply to the maintainability evaluation. Table 1 outlines the sections of this document that must be reviewed by each member of the evaluation team prior to an evaluation.

Table 1. Review Guidelines.

	CHAPTERS						ATTACHMENTS							
ROLE	1	2	3	4	5	6	1	2	3	4	5	6	7	8
STM	X	X	X	X	X	X							X	
DSE	X	X	X			X							X	
Moderator	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Evaluator	X				X	X	X	X	X	X		X		X

1.1. Software Supportability.

1.1.1. The supportability of software is an important aspect of the operational suitability of many systems. Software almost always changes after initial delivery to the customer. Software must be modified to correct errors, enhance system capabilities, and adapt software to be compatible with other system changes. The amount of effort necessary to make modifications to software is affected by its development process, characteristics of the software product, and the support environment (see figure 1). This effort—called software supportability—is often a dominant factor in a system's life-cycle cost and responsiveness to changing mission requirements. Consequently, a software system that cannot continue to evolve to meet user requirements at reasonable cost is not suitable.

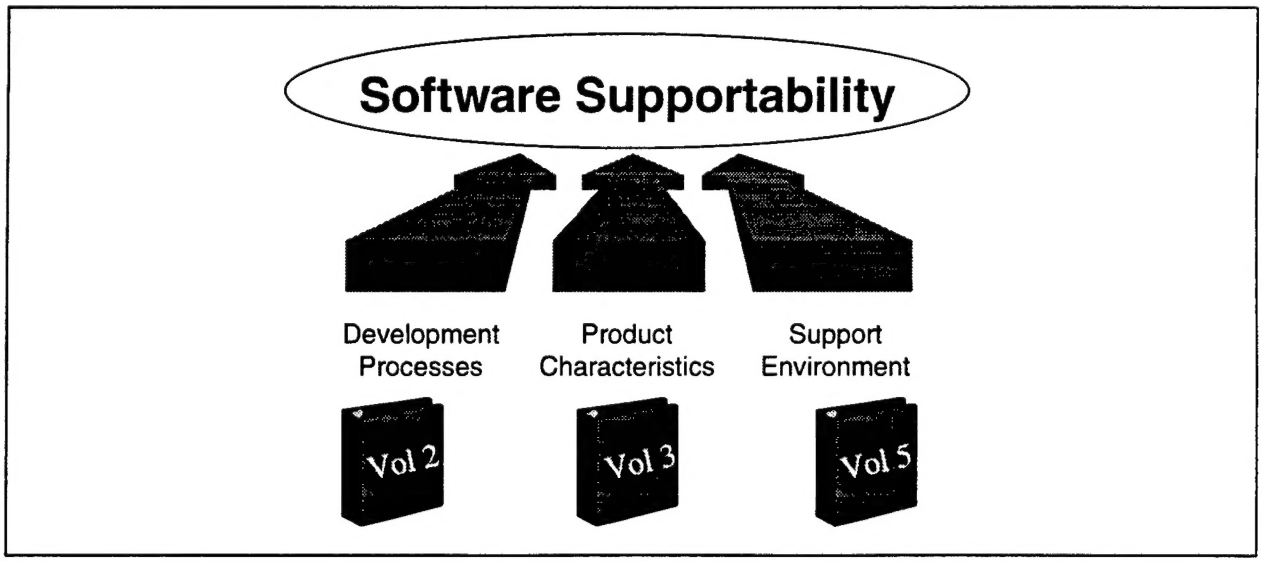

Figure 1. Elements of Software Supportability.

1.1.2. Ideally, a software supportability evaluation would directly measure the effort required to make changes to software—similar to the way system "maintainability" is evaluated during OT&E[1]. Typically system maintainability is measured by recording the time and effort involved in restoring the system to a "mission ready" state after a failure. The range and mean value of the time required to complete maintenance actions during an operational test period are used as a measure of the system's "maintainability." However, there are a number of problems with using this approach to measure software supportability during OT&E, including (1) software maintenance is not performed in real-time, but at a depot-level facility that may not be fully operational during OT&E, (2) software maintenance performed before and during OT&E is usually not operationally representative, and (3) DoD policy prohibits the implementation of software changes during OT&E. Consequently, AFOTEC measures software supportability by evaluating various characteristics of the development process, the software product, and the support environment which together make software supportable at reasonable cost.

1.1.3. Software supportability is typically included in OT&E test concepts as a measure of effectiveness (MOE) under system suitability. AFI 10-602 directs ORD writers to incorporate integrated logistics support (ILS) elements into every ORD. Whether or not the ORD includes these ILS elements, the test support group should establish at least one suitability COI to assess ILS elements, and software supportability is one of these elements. Figure 2 provides an example of a COI structure that includes a software supportability MOE. While this structure is not applicable to every program, a similar measure should be included in the test and evaluation master plan (TEMP) of each program.

COI X System Suitability. (or equivalent COI)

MOE X-Y Software Supportability. *A composite rating of the processes used to develop the software, the maintainability of the software products, and the processes and resources established to support the software system once fielded. This measure represents an overall assessment of the ability of the support agency to sustain a software product that will continue to meet the mission needs of the system.*

MOP X-Y-1 Software Support Life Cycle Process.

MOP X-Y-2 Software Maintainability

MOP X-Y-3 Software Support Resources

Figure 2. Software Supportability MOE.

1.1.4. The Software Supportability MOE is further decomposed into measures of performance for each of the three elements of software supportability. The evaluation method outlined in this guide can be used to resolve the software maintainability MOP. Other volumes of this pamphlet that address software supportability MOPs include the *Software Support Life Cycle Process Evaluation Guide* (volume 2) and the *Software Support Resources Evaluation Guide* (volume 5).

1.2. Software Maintainability Evaluation Overview.

1.2.1. What is "Software Maintainability?"

Software maintainability is defined as a measure of the ease with which a software product can be understood, modified, and tested.

[1] Here we are using the definition of maintainability from DoDI 5000.2, Part 15, "The ability of an item to be retained in or restored to a specified condition when maintenance is perfomed by personnel having specified skill levels, using prescribed procedures and resources, at each prescribed level of maintenance and repair."

This evaluation focuses on characteristics of the software product that help make it maintainable. Software maintainability is defined as a measure of the ease with which a software product can be understood, modified, and tested. The difficulty (or ease) of modifying software is determined mainly by the quality of the original software product—its architecture, the workmanship exercised by its builders, and the quality of its development environment. In particular, the software architecture is as important as the hardware architecture of a system; it must have well-defined protocols, interfaces, and properly structured and documented source code.[2]

1.2.2. What are the Goals of a Maintainability Evaluation? The OT&E goals of the maintainability evaluation are (1) to measure and report a maintainability rating to help resolve a measure of effectiveness for software supportability, (2) to identify deficiencies to the acquirer and developer to facilitate product improvements, and (3) to help the Software Support Activity (SSA)—the organization responsible for post-deployment software support—gauge the resources required to maintain the software.

1.2.3 How is Software Maintainability Evaluated?

1.2.3.1. The evaluation method described in this guide determines the presence (or absence) of desirable characteristics of a software product important for successful software maintenance. Typically a team of experienced maintainers from the support agency, led by an AFOTEC moderator, complete standardized questionnaires on the product's documentation and a representative sample of the source code. A representative sample is necessary so that the results of evaluating a sample of source code can be used to rate the entire system. The product's implementation (design) is evaluated by the moderator (prior to or following the team evaluation), not the evaluation team, because many of these questions require in-depth investigation not possible during the short duration of a maintainability evaluation.

1.2.3.2. For each question, the software (source code and documentation) is rated on a scale of 1 (bad) to 6 (good). The average evaluator score for each question is aggregated to develop an overall maintainability rating for documentation, source code, and implementation. Ratings below the AFOTEC threshold of 3.0 indicate unacceptable (poor) maintainability. Ratings in the range between 3.0 and 4.0 indicate marginal maintainability, and ratings at 4.0 or above indicate acceptable (good) maintainability (figure 3).

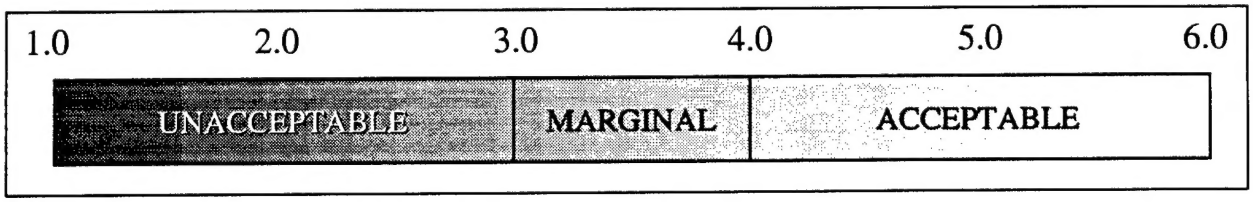

Figure 3. Maintainability Rating Scale.

Each major software component or Computer Software Configuration Item (CSCI) within a system is separately evaluated for maintainability. However, the scores for separate CSCIs are sometimes aggregated together if the acquisition decision-maker needs the scores grouped in a specific way to better support acquisition decisions, or convey problem areas to the developer.

1.2.4. When are maintainability evaluations conducted?

A software product is considered to be "production representative" for purposes of a maintainability evaluation when source code and documentation are in near final form, and no major changes are expected that might alter the maintainability characteristics prior to fielding the system.

[2] USAF Scientific Advisory Board, Report of the Ad Hoc Committee on Post Deployment Software Support, December 1990.

In general, maintainability evaluations should be scheduled as soon as possible after each CSCI is available in "production representative" form. Figure 4 shows a typical schedule for a system with six CSCIs to be evaluated. It is not necessary to wait until the exact version of the product that will be used during dedicated OT&E is available, but the software should not be expected to undergo major changes that might alter its maintainability characteristics prior to fielding the system. Ideally, evaluations are scheduled to provide the acquirer and developer with feedback early enough in the software development process to implement improvements. If the evaluations are completed late in the program, they may still be of value to the acquisition decision-maker, but it will be difficult to implement improvements and the corrections will be much more expensive.

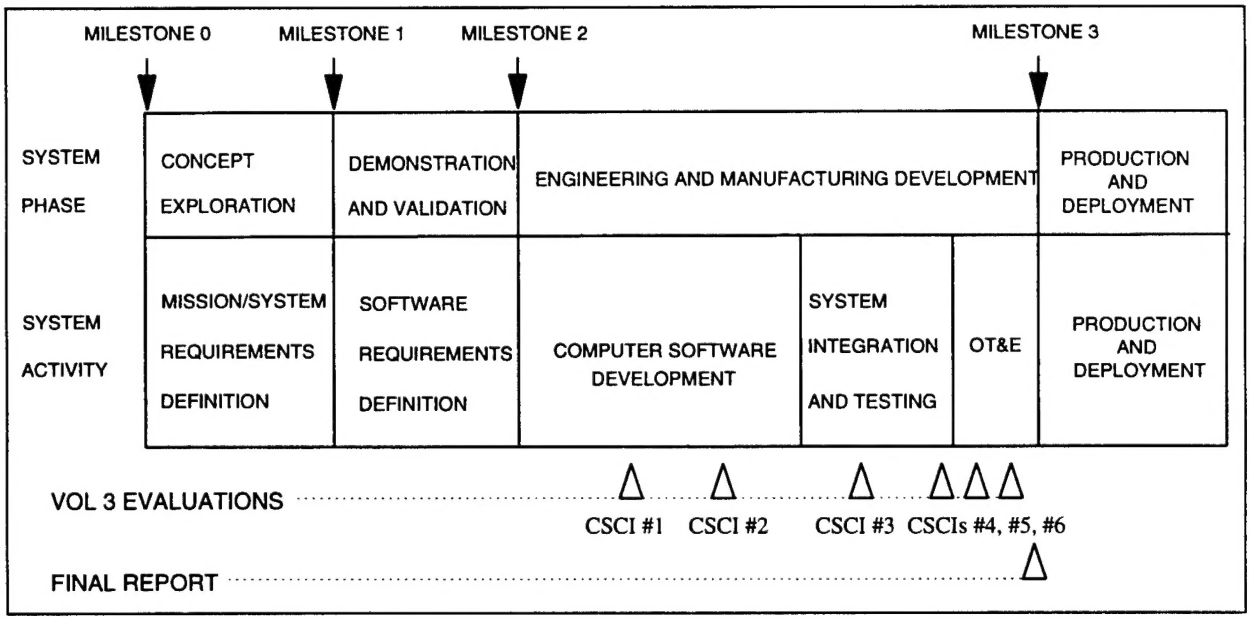

Figure 4. Volume 3 Evaluation Timeline.

1.2.5 What are the primary responsibilities of the STM and DSE? The AFOTEC software test manager (STM) is responsible for developing a software test concept for assigned OT&E programs. The software test concept defines which evaluations will be completed, what software will be evaluated, when it will be evaluated, and what resources are required to complete the evaluations. If a DSE is already assigned to the test program, the STM and DSE will work together to determine what software evaluations will be included in the test concept. The DSE is responsible for conducting software evaluations according to an approved test plan and the guidelines of AFOTEC Pamphlet 99-102. Typically the DSE will perform the detailed planning for each maintainability evaluation, moderate the evaluations, and report the results.

1.2.6. Where is the Software Evaluation Plan Documented? The software test concept is outlined in the test and evaluation master plan (TEMP) and OT&E test plan, and should be further described in detailed test procedures (either formal detailed test procedures as outlined in AFOTEC Instruction 99-101, or informal detailed test procedures maintained by the DSE). The AFOTEC STM identifies the resources required to complete the software evaluations in the test resources plan (TRP).

2. Elements of Software Maintainability. The method for evaluating software maintainability is based on the use of closed-form questionnaires with optional (but important) written comments. The questionnaires are designed to focus on the elements of software maintainability shown in figure 5. This hierarchical evaluation structure enables an evaluator to identify potential maintainability problems at various levels: category (documentation, source code, and implementation), characteristic (e.g., modularity or consistency), or component (document, CSU, or module). These elements of maintainability are involved in almost all software systems, and are generally applicable to each software maintainability evaluation.

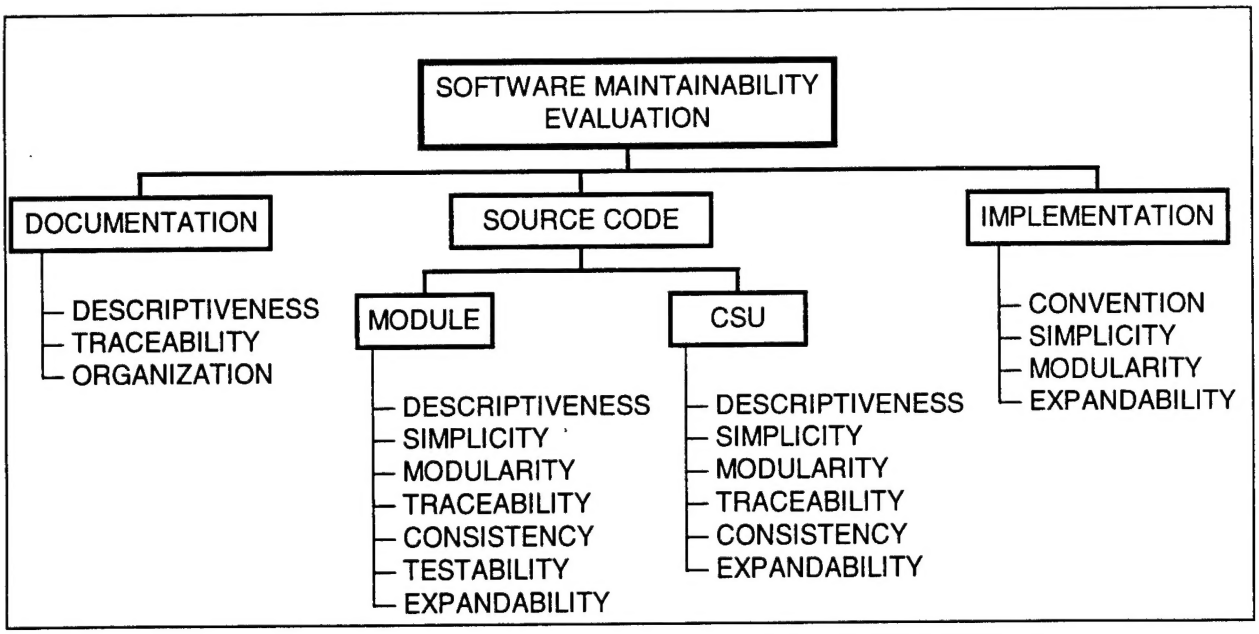

Figure 5. Software Maintainability Evaluation Categories and Characteristics.

2.1. Software Categories. For each software system, there are related categories of software product information that affect maintainability. Most software systems consist of a set of computer instructions and data structured into programs (source code files), and associated documentation on the requirements, design, implementation, test, support, and operation of those programs. Maintainers depend upon information from each category of a software product—documentation, source listings, and implementation—to help understand, modify, and test the system. Each category is evaluated separately using a questionnaire designed to focus on characteristics that affect the maintainability of the software.

2.2. Software Maintainability Characteristics. The maintainability of a software system is determined by evaluating the characteristics shown in figure 5. Each question in the questionnaires evaluates an attribute of one of these characteristics.

2.2.1. Software Documentation.

2.2.1.1. Maintainers depend upon a complete and accurate set of documentation to (1) identify system, software, and interface requirements, (2) describe software and interface designs, (3) prescribe design and coding standards, and (4) describe how to test the software.

2.2.1.2. The documentation used in this evaluation consists of requirements, design, test, and support information. These documents may have a variety of forms depending upon the software standards (e.g., MIL-STD-498) and engineering environments (e.g., CASE tools) used to develop and maintain the system, and tailoring specified within each acquisition program. The documentation evaluated should be in final or final draft form and should not be expected to undergo major changes prior to operational use. Use only the documentation that will be available to the software maintenance organization, and in the form it will be used by maintainers.

2.2.1.3. The documents are evaluated for descriptiveness, organization and traceability. See attachment 2 for a description of the documentation characteristics evaluated.

2.2.2. Software Source Code.

2.2.2.1. Software source listings (source code) are the program code in the source language (e.g., FORTRAN, Ada, assembly language). The source listings represent the program as implemented, in contrast to the documentation,

which usually represents the intended program design or implementation plan. In essence, source listings are also a form of program documentation, but for this maintainability evaluation a distinction is made.

2.2.2.2. The source listings used for the evaluation should not be expected to undergo major changes prior to operational use. The listings evaluated should also be compiled whenever possible to include compiler warnings and other useful information.

2.2.2.3. The source listings evaluation consists of two separate evaluations at different levels of implementation (see figure 6). A sample of modules is evaluated using the module-level questionnaire. A sample of computer software units (CSU) are also evaluated. The CSU questionnaire provides additional information at a level higher than the module questionnaire. The results of each evaluation are reported separately and give two perspectives on the maintainability of the software source listings for the CSCI. If the CSU level and the module level are determined to be the same, the CSU-level questionnaire is not used.

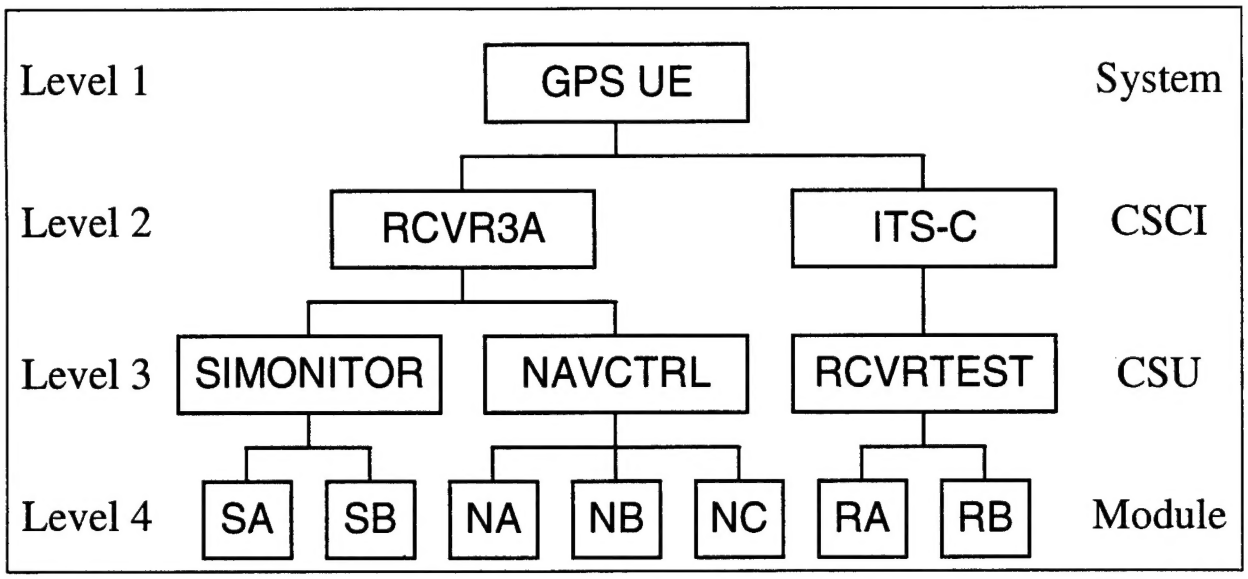

Figure 6. Software Evaluation Hierarchy.

2.2.2.3.1. Module Source Listings. This questionnaire determines how well the design has been implemented at the module level. A module is defined as the smallest callable component of source code. Examples of modules are:

- Ada: subprogram (procedure or function)
- FORTRAN: subroutine
- C: function
- JOVIAL: subroutine (procedure or function)

2.2.2.3.2. CSU Source Listings. This questionnaire determines how well the source code has been broken up into CSUs or packages. A CSU is a component part of a software product composed of one or more modules. The moderator should define the CSU at a level of modularity most appropriate for the system being evaluated. As a rule of thumb, the CSU level should be set at the lowest level at which a software development file (SDF) is maintained. Some examples of CSUs are:

- Ada: package
- C: *.C file plus any associated *.H files

2.2.2.4. The source listings are evaluated for descriptiveness, simplicity, modularity, traceability, consistency, testability, and expandability. See attachments 3 and 4 for a description of the source code characteristics evaluated.

2.2.3. Software Implementation.

The moderator reviews the questionnaire and gathers the supporting information prior to the team evaluation.

2.2.3.1. Implementation is the result of representing the software design in the documentation and the source listings. Decisions made as to how the software design is implemented, such as the language used, greatly affect the maintainability of the software.

2.2.3.2. The implementation evaluation consists of a separate questionnaire which examines aspects of both external documentation and source listings. The implementation questionnaire results yield an overall evaluation of how well the program was designed and implemented. Extensive knowledge about the system is required in order to accurately complete this evaluation. The moderator reviews the questionnaire and gathers the supporting information prior to the team evaluation. An extensive effort may be required to obtain the information needed to accurately answer the implementation questions. After compiling this information, the moderator should assist remaining evaluators to ensure the implementation questions are answered accurately.

2.2.3.3. The implementation is evaluated for convention, simplicity, modularity, and expandability. See attachment 5 for a description of the implementation characteristics evaluated.

3. Maintainability Evaluation Phases. The software maintainability evaluation involves five distinct phases: test concept development, test planning, calibration, assessment, and analysis/reporting, as shown in table 2.

3.1. Test Concept Development Phase. During the software test concept phase, the STM develops a plan for evaluating the supportability of software. This supportability evaluation almost always includes software maintainability evaluations. The test concept development phase is described in detail in AFOTECP 99-102, volume 1, *Management of Software Operational Test and Evaluation.*

3.2. Evaluation Planning Phase. During the planning phase, the STM and/or the DSE will set up an evaluation team consisting of at least five qualified evaluators. The program/module hierarchy is established and the source code is sent to HQ AFOTEC/SAS where a sample is generated. Finally, an evaluation location is selected and detailed preparations are made. Figure 7 details this process by showing what activities are required beginning approximately 3.5 months prior to a maintainability evaluation. Actual timing will vary from program to program.

3.2.1. Finding Evaluators.

Evaluators should NOT be development contractors, SPO personnel, or anyone perceived to have a bias (for or against) the system being evaluated.

3.2.1.1. Ideally, a team of five or more evaluators from the SSA is used to conduct the evaluation, but the availability of SSA evaluators may be limited or the cost may be prohibitive. AFOTEC is responsible for funding all reimbursable resources required to support operational testing, which includes both TDY expenses and the salaries/benefits of SSA evaluators. Other sources of evaluators include the operational test team, AFOTEC general support contractors, and HQ AFOTEC/SAS. Evaluators should have experience in software development or maintenance and should have completed at least an introductory-level course in the languages in which the source code was developed. Evaluators should NOT be development contractors, SPO personnel, or anyone perceived to have a bias (for or against) the system being evaluated. See section 6.6 for information on conducting a maintainability inspection when fewer than five evaluators are available.

Table 2. Summary of Maintainability Evaluation Procedures.

Test Concept Development	**Software Test Manager (STM) Duties (for the OT&E program)** ❑ Decide if a software supportability evaluation is needed ❑ Develop an aggregation strategy for rating overall software supportability ❑ Document the test concept in the TEMP, OT&E test plan ❑ Decide what CSCIs will be evaluated for maintainability ❑ Develop an overall schedule for completing maintainability evaluations ❑ Determine the primary source of evaluators ❑ Estimate resource requirements and document in the TRP ❑ Identify any unique methods, tools, or training needed	
Test Planning	**Deputy for Software Evaluation (DSE) Duties (for each evaluation)** ❑ Decide which CSCI will be evaluated during this evaluation ❑ Develop specific schedule for this evaluation ❑ Obtain evaluation materials ❑ Determine the hierarchy of the CSCI to be evaluated ❑ Obtain a representative sample of source code ❑ Select evaluators/arrange for funding ❑ Prepare evaluator briefing ❑ Brief program personnel on the evaluation approach/requirements ❑ Determine the distribution of the report	
Calibration	**Moderator Duties** ❑ Present evaluator briefing ❑ Thoroughly discuss each question ❑ Review completed questionnaires for width and outlier flags ❑ Resolve any misunderstandings or disputes on question interpretation ❑ Debrief the evaluators	**Evaluator Duties** ❑ Complete one documentation questionnaire ❑ Complete one module-level questionnaire ❑ Complete one CSU-level questionnaire ❑ Complete one implementation questionnaire ❑ Update calibration questionnaires
Assessment	**Moderator Duties** ❑ Analyze completed questionnaires for width and outlier flags ❑ Resolve any misunderstandings or disputes on question interpretation ❑ Encourage thorough discussion of issues ❑ Encourage written comments ❑ Complete implementation questionnaire before team evaluation	**Evaluator Duties** ❑ Complete questionnaires ❑ Update documentation questionnaire mid-way through source code evaluation ❑ Write down all comments and general impressions
Analysis and Reporting	**Moderator Duties** ❑ Log scores in WINQAS ❑ Generate appropriate WINQAS reports ❑ Review evaluator comments ❑ Complete a maintainability evaluation activity report ❑ Send final WINQAS files to HQ AFOTEC/SAS	**STM or DSE Duties** ❑ Review/approve the maintainability evaluation activity report (if required) ❑ Develop inputs to the OT&E report

3.2.2. Selecting a Source Code Sample.

The objective of sample selection is to pick the smallest sample possible that is still representative of the system.

3.2.2.1. It is usually impractical to evaluate all of the source code within a CSCI. Thus, a sample of source code is required. The ability to generalize the results of the source evaluation from this sample to the source code population requires that the sample be representative of the entire population. The objective of sample selection is to pick the smallest sample possible that is representative of the system. Whenever possible, AFOTEC's Automated

Software Evaluation Tools System (ASETS) should be used to select a statistically representative sample of source code for evaluation.

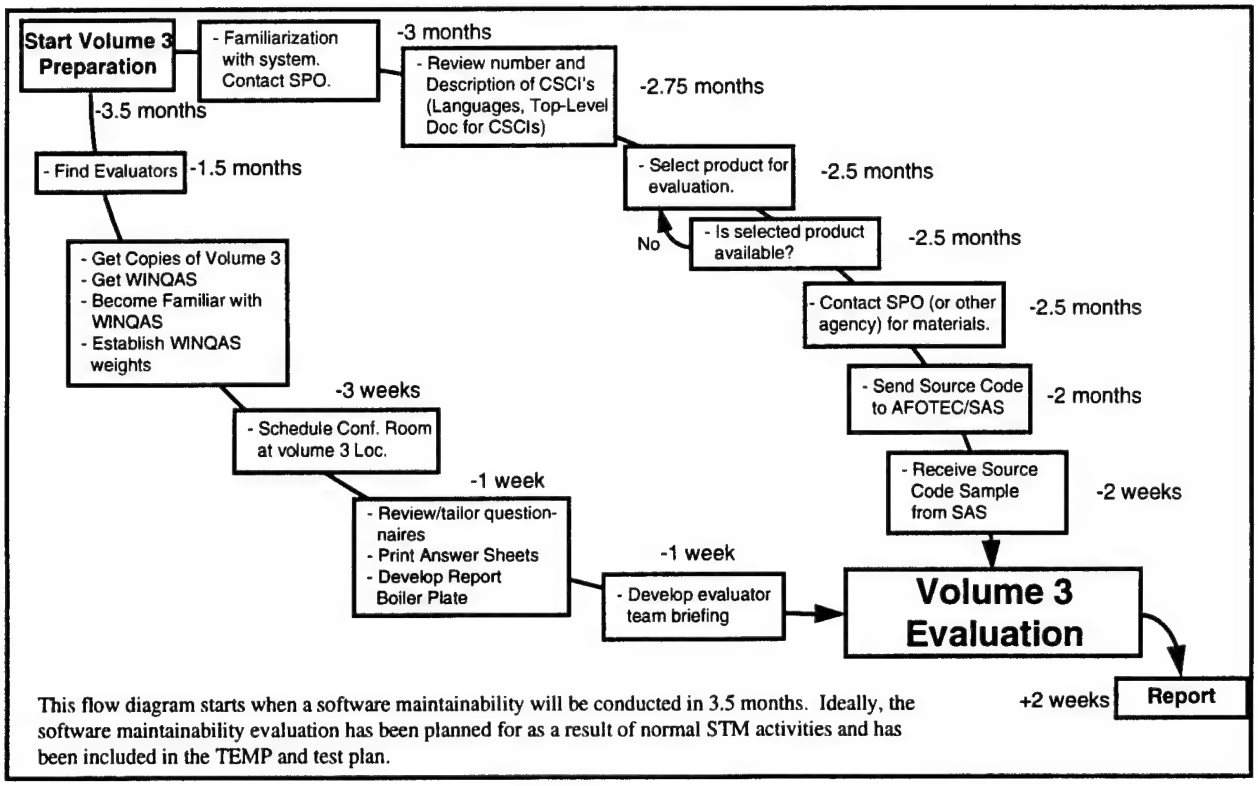

Figure 7. Maintainability Evaluation Planning Flow.

3.2.2.2. During early planning for software evaluation, STMs/DSEs should review the ability of ASETS to support their program, and determine the feasibility and cost-effectiveness of using ASETS. Factors to be considered include the languages used (ASETS currently supports five languages—Ada, C, JOVIAL, FORTRAN, and COBOL), the classification of the code, government data rights to the source code, the volume of code anticipated, the evaluation schedule, and whether the source code can be provided in the proper format at reasonable cost. If an upgrade to ASETS is necessary to support a program, then HQ AFOTEC/SAS will make every attempt to ensure capabilities are available when needed. Since ASETS is undergoing continuous improvements, contact HQ AFOTEC/SAS for the latest information on using ASETS to select a source code sample

3.2.2.3. When ASETS cannot be used, a sample must be selected manually. The sample evaluated includes both a sample of modules and a sample of CSUs. The following paragraphs further describe ASETS and manual sample selection (figure 8).

3.2.2.4. Selecting a Sample Using ASETS.

3.2.2.4.1. The first step in selecting a source code sample using ASETS is to provide source code and a completed ASETS evaluation request form (attachment 8) to HQ AFOTEC/SAS. From this source code, ASETS will select a sample of modules that mimics as closely as possible certain quantitative characteristics of the entire system. This sample is known as the representative sample. In general, the results of evaluations conducted on an ASETS-generated sample will more accurately reflect the maintainability of the entire CSCI than evaluations conducted on a purely random sample of the same size. ASETS can also be used to reduce the sample size to as low as 2 percent of the original source code (or 10 modules in systems with fewer than 500 source modules).

Once a representative sample is selected, it should not be altered, since that will invalidate the sample.

3.2.2.4.2. Once the representative sample is selected, it should not be altered since that will invalidate the sample. Modules outside the population processed by ASETS (e.g., assembly language modules) are treated separately during sample selection, and then combined with the ASETS sample. Special cases involving representative samples can be accommodated if identified ahead of time.

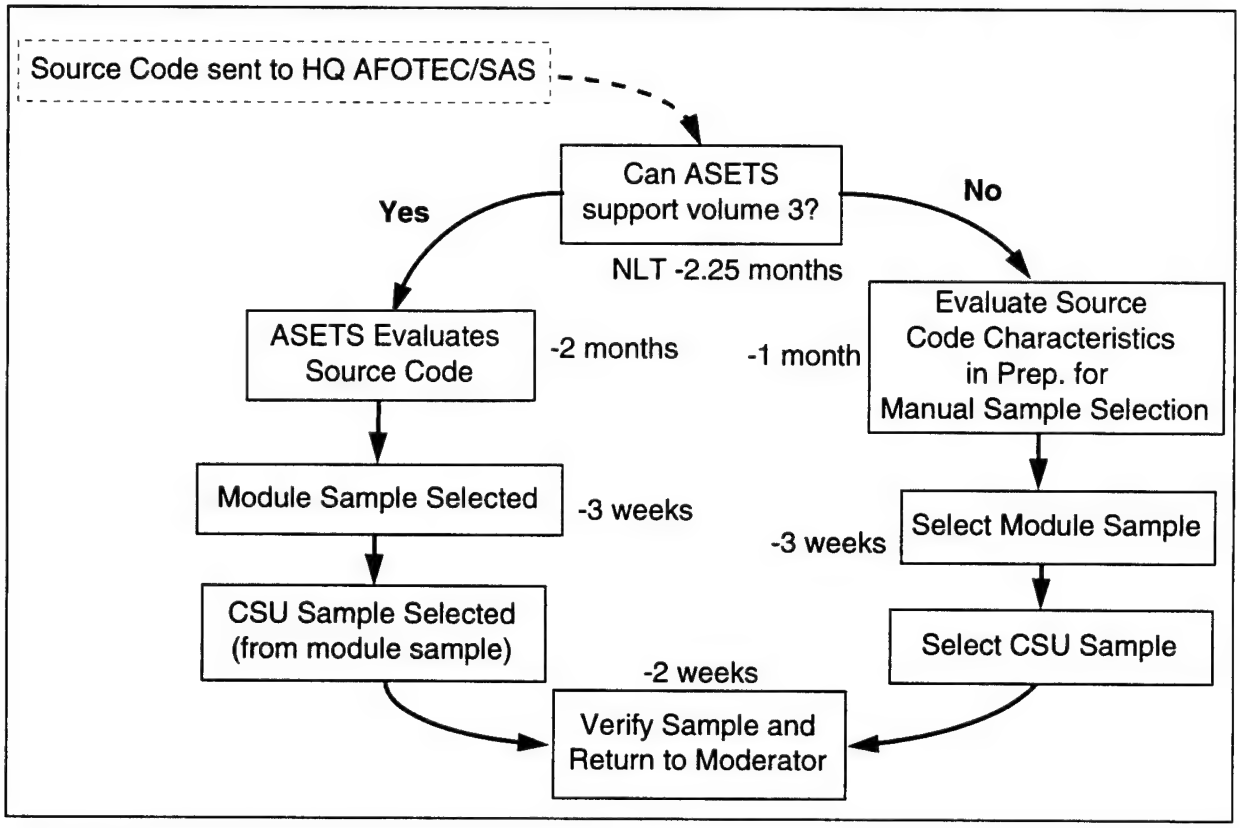

Figure 8. HQ AFOTEC/SAS Sample Selection Process.

3.2.2.4.3. After a module sample has been provided by ASETS, the CSU sample is obtained by selecting the CSU-level components associated with the module sample. If two or more of the modules in the sample are included in the same CSU, that CSU is evaluated only once. In the case where the module is the same as the CSU-level, the CSU questionnaire will be omitted.

3.2.2.4.4. Prior to evaluating the source code, the STM must compare the characteristics of the representative sample to those of the entire system and verify the quality of the representative sample.

3.2.2.5. Selecting a Sample Manually. A manually selected sample must include at least 10 percent of the source code components to retain the same statistical validity of a smaller ASETS-generated representative sample.

3.2.2.5.1. The process for manually selecting a sample is closely tied to the organization of the software components. Typically, a CSU is a computer file. In this case, the evaluator can select 10 percent of the files for the CSU evaluation using the partitioning strategy outlined in section 3.2.2.5.3. Next, count the modules within the CSU sample. This number can be used to estimate the number of modules within the entire CSCI. Ten percent of the estimated number of modules should then be selected (from throughout the entire CSCI) for module-level evaluation. See figure 9 for an example of this sample selection process.

3.2.2.5.2. If modules are computer files for a system, 10 percent of these files should be selected for a module-level evaluation using the partitioning strategy outlined in section 3.2.2.5.3. Once this module sample is selected, the CSU sample is obtained by selecting the CSU-level components associated with the module sample. If two or more of the

modules in the sample are included in the same CSU, that CSU is evaluated only once. In the case where the module is the same as the CSU-level, the CSU questionnaire will be omitted.

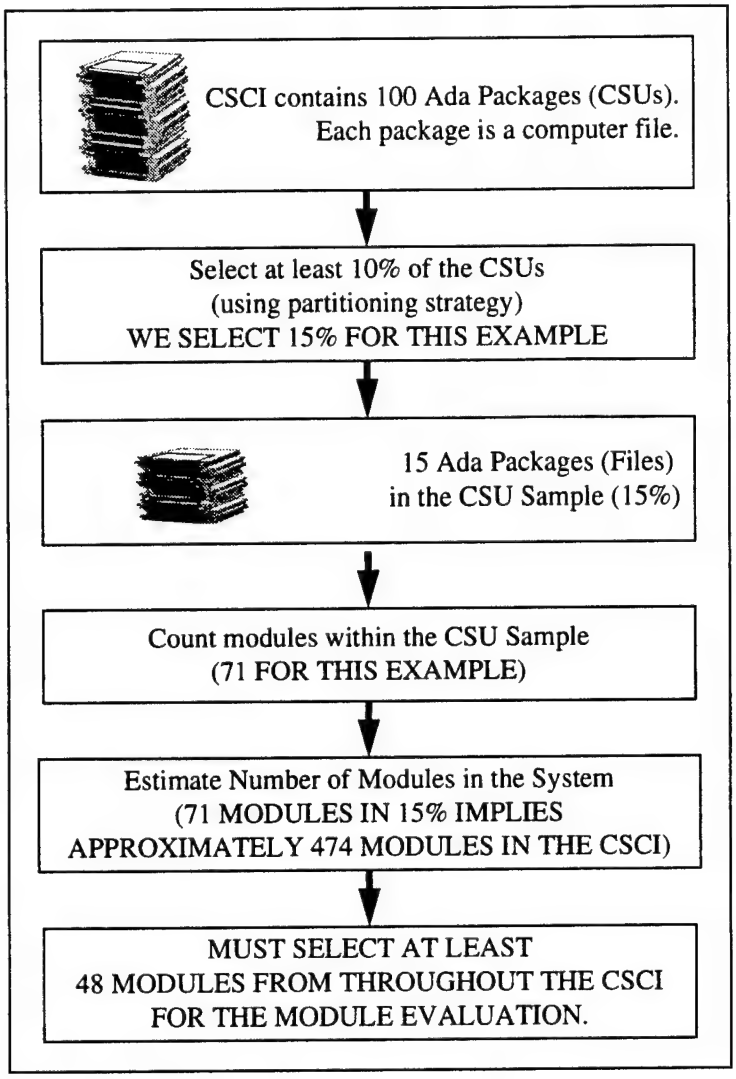

Figure 9. Manual Sample Selection Example.

3.2.2.5.3. When manually selecting a sample, partition the population of components (modules or CSUs) using all available information about the source code. The objective of partitioning is to group the components by some common characteristics, and then select components from each of group to obtain a more representative sample.

For example, if size (lines of code or even file size) is known, then the population can be partitioned into groups of different sizes. Other examples of partitioning characteristics include:

- Language
- Type of CSU (driver, input/output, utility, etc.)
- Functional Area or System Function
- Author

After the population has been partitioned, select a random sample from each group. In the unlikely event no partitioning information is available, then randomly select at least 10 percent of the components.

3.2.3. Selecting an Evaluation Facility. Ensure that adequate facilities are available for the evaluation. A large conference room away from the normal work area is a good location. It is often helpful to conduct the evaluation at the SSA or development contractor facility so that questions about the software engineering environment and/or documentation can be easily answered. This is especially important when the documentation is embedded in CASE tools.

3.2.4. Obtaining the Evaluation Materials.

3.2.4.1. The most important evaluation material is, of course, the documentation and source listings to be evaluated. Table 3 provides a list of documents that may be available depending on which software development standard has been used. Since the contractual requirements for software deliverables are often tailored, all of the specific documents listed in table 3 may not be available. However, material which fulfills the intent of each document should be available to the evaluators. Obtain copies of all information that will be available to the software maintainers for maintenance activities.

Table 3. Recommended Documentation List.

INFORMATION NEEDED	EXAMPLE DOCUMENTS
System and software specifications	Software Requirements Specification (SRS) Interface Requirements Specification (IRS) System/Subsystem Specification (SSS)
Software design descriptions	Software Design Description (SDD) Database Design Description (DBDD) Interface Design Description (IDD) System/Subsystem Design Description (SSDD)
Standards and conventions	Software Development Plan (SDP) Programmers Manual Style Guide
Support documents (These documents have information about the support environment)	Computer Resource Integrated Support Document (CRISD) Computer Resource Integrated Support Plan (CRISP) Computer Resources Life Cycle Management Plan (CRLCMP)
Version description	Software Version Description (SVD) Version Description Document (VDD)
Software test descriptions and results, including sample test data	Software Test Plan (STP) Software Test Description (STD) Software Test Report (STR)
Maintenance manual, including how to use the support software—compilers, editors, etc.	Computer Programming Manual (CPM)
Interface descriptions	Interface Design Description (IDD) Interface Control Document (ICD)
Software analysis/design information (***NOTE***: This information may be included in design documents)	Program Design Language (PDL) Flowcharts HIPOs Booch/Buhr diagrams Data Flow Diagrams (DFD)
Timing and sizing reports	CDR PDR Slides Metrics Reports, etc.
The source code listings for the CSUs/modules evaluated	
Software development files (SDF) for the CSUs/modules evaluated	
Any other deliverable or supporting documentation that will be available to the software maintenance organization	Other MIL-STD-498 Documents Development Information, etc.

3.2.4.2. Ensure that enough copies of this volume are available (one for each evaluator). If additional copies are needed, they should be ordered in sufficient time or copied prior to the evaluation.

3.2.4.3. Standard answer sheets are provided at attachment 8. Electronic versions of these answer sheets are available from HQ AFOTEC/SAS. While there is no requirement to use a standard answer sheet, the moderator must ensure that question scores and comments are recorded for later analysis.

3.2.4.4. The moderator should obtain version 2.0 of the Windows Questionnaire Analysis System (WINQAS) from HQ AFOTEC/SAS. This evaluation tool is used to record questionnaire scores, analyze the data, and generate charts used to summarize the results. WINQAS applies the appropriate weightings and returns documentation, source code, and implementation scores. Do not use the Field Questionnaire Analysis System (FQAS), the predecessor to WINQAS, with this volume.

3.3. Calibration Phase.

3.3.1. A calibration (lockstep answering of the questionnaire) is performed for each evaluation. The documentation and the first of each type of source code questionnaire is completed as a calibration run.

3.3.2. The function of the calibration phase is to improve the reliability of the evaluation by confirming that each evaluator has a clear understanding of the questions and the response (scoring) guidelines. Each evaluator completes one documentation questionnaire, one module questionnaire, and one CSU questionnaire in a trial or calibration evaluation guided by the moderator. Normally, the documentation calibration is the first evaluation performed. Next, the source code calibration is performed. Then all the remaining source code evaluations are completed.

3.3.3. During the calibration, the moderator should lead a discussion of each question and use WINQAS to help identify disagreement among the evaluators on how to interpret or score questions. WINQAS analyzes the distribution of scores for each question. When the scores do not generally follow a normal distribution, WINQAS flags the question with an "outlier" or "width" problem in the distribution of scores. These flags indicate that one or more of the evaluators may have a significant disagreement with the others on how to interpret the question. Evaluators should have a uniform interpretation of how each question applies to the system, but should never be forced to change a score.

Evaluators should have a uniform interpretation of how each question applies to the system, but should NEVER be forced to change a score.

3.4. Assessment Phase. In the assessment phase, the evaluation team updates their calibration test questionnaires based on the results of the calibration debriefing. The team then completes the remainder of their assigned source listing questionnaires. Following the completion of each answer sheet, the moderator should review the answers using WINQAS to ensure that the evaluation team had a uniform understanding of how each question applies. To preserve the statistical validity of the evaluation, it is important that the distribution of answers be tightly grouped about the mean score, but evaluators should NEVER be required to change a score. The evaluation team should continue to discuss any questions with "outlier" scores.

3.5. Analysis and Reporting Phase. In the analysis and reporting phase, the moderator accomplishes the data processing of the questionnaire data using WINQAS. The statistical summaries are used as the basis for preparing the maintainability evaluation activity report. Section 4.7 provides more details on writing a maintainability evaluation report.

4. Moderator Guidelines.

Being a moderator is more difficult than being an evaluator. In addition to the fatigue of the evaluation, the moderator must plan the evaluation, keep the discussions focused, analyze the results, and write the evaluation report.

4.1. Sample Evaluation Schedule.

4.1.1. The moderator is free to schedule the evaluation in any reasonable manner. Table 4 provides a typical schedule for a 1-week evaluation. The moderator should be prepared to spend additional time prior to the evaluation to complete the implementation questionnaire and time afterward to outbrief the SPO or contractor representative on the evaluation. On some systems, it may not be possible to complete the evaluation in 1 week. The purpose of this

schedule is to illustrate what categories should be evaluated and when. The source code evaluation includes both the module and CSU evaluations.

4.1.2. The rationale for the "quick look" documentation evaluation on day 1 and documentation evaluation on day 3 is the evaluators will become increasingly familiar with the documentation and may be able to provide more accurate answers and comments on day 3. The day 1 "quick look" evaluation is necessary to familiarize the evaluators with the documentation for the source code evaluation. As the evaluators become familiar with the system, the questionnaire, and the response scale, they will tend to move faster as the week progresses.

Table 4. Sample Evaluation Schedule.

TIME	MONDAY	TUESDAY	WEDNESDAY	THURSDAY	FRIDAY
AM	Intro Briefing Doc "quick look" Eval	Source Code Evaluation	Documentation Evaluation	Source Code Evaluation	Source code (if needed)
PM	Source Code Evaluation	Source Code Evaluation	Source Code Evaluation	Implementation Evaluation	Implementation Evaluation (if needed)

4.1.3. The rationale for two sessions for implementation is because this evaluation may involve discussion among the team or time to obtain additional information. Since the moderator completes the implementation questionnaire prior to the team evaluation, he/she may be able to provide any additional information. The team should complete the entire questionnaire in the first session and review the answers in the second session.

4.2. Question Response Scale Guidelines

4.2.1. Normal Responses. The responses 1 to 6 show the extent to which the evaluator disagrees or agrees with the question statement. The meaning of each response is provided in table 5. The responses 4, 5, and 6 are used when the evaluator agrees that the attribute addressed by the question is present in the software product. The selection of 4, 5, or 6 is based upon each evaluators assessment of how well the attribute has been implemented to facilitate maintenance of the software. The responses 1, 2, and 3 are used when the evaluator disagrees that the attribute addressed by the question is present in the software product. The selection of 1, 2, or 3 is based upon each evaluator's assessment of the impact of the deficiency on the maintainability of the product. Depending on the application area and the type of question, these responses can be interpreted differently.

4.2.2. N/A Response. Occasionally, the evaluator may find it difficult to answer a question because it "doesn't seem to apply" to the system being evaluated. This situation may arise when the software requirements do not involve the need for a certain maintainability attribute, or the intrinsic nature of the software product (e.g., language or engineering environment used) eliminates the need for such an attribute. Some questions imply the existence of a particular item (e.g., preface block in source listing) but the question asks specifically about the content of the item. If the item does not exist, then this probably indicates a maintainability deficiency in the software. In this case, the evaluators should show disagreement with the question (responses 1, 2, or 3). It is rare that some application for a question cannot be found. Should the moderator, with the consensus of the evaluation team, determine that a question has no application to the system being evaluated, then the N/A response is appropriate. *To be a valid response, every evaluator must mark only N/A.* The question will then be dropped from the automated analysis. It is not appropriate for a single evaluator or subset of evaluators to answer N/A. The evaluation moderator will make the final call in the event the evaluators don't reach consensus. In some cases, the moderator may decide that questions are not applicable prior to the evaluation. In this case, the evaluation team will disregard any question marked N/A by the moderator.

Table 5. Question Response Scale.

RESPONSE	MEANING	EXPLANATION
6	Decidedly Agree	This response indicates the product being evaluated is excellent with respect to the attribute addressed.
5	Moderately Agree	This response indicates the product being evaluated is very good with respect to the attribute addressed.
4	Somewhat Agree	This response indicates the product being evaluated is barely acceptable with respect to the attribute addressed.
3	Somewhat Disagree	This response indicates the product being evaluated is unacceptable with respect to the attribute addressed.
2	Moderately Disagree	This response indicates the product being evaluated is very poor with respect to the attribute addressed.
1	Decidedly Disagree	This response indicates the product being evaluated is terrible with respect to the attribute addressed.
N/A	Not Applicable	The statement does not apply to this software product. *Do not use without moderator approval.*

4.2.3. Special Response Instructions. Each question contains a field for special response (scoring) instructions. These instructions direct the evaluator to a specific score under certain conditions. For most questions, this field is set to "None," meaning the evaluator is free to choose the most appropriate response from the full range of values based upon the response scale guidelines. In some cases, the evaluation team is directed to avoid certain answers. When special response instructions are provided, they will have one or more of the following goals:

- To prevent rewarding or penalizing the system for an attribute which is not necessary.
- To provide additional guidance on the use of the "N/A" response.
- To prevent rewarding the system for an attribute that is in the "meets standard" category meaning its nonexistence is bad, but its existence should be expected.

4.3. Use of Metrics Data During the Evaluation. Occasionally some metrics data related to a maintainability characteristic may be available. In particular, if ASETS has been used to select a source code sample, then a condensed report of metric information will be available. The moderator must decide how to use this information. One approach is to use the data to verify the sample is valid, and not use it for any other purpose. Another approach is to provide the data to each evaluator, along with a brief definition of each metric, and let him/her decide how to use it to support the evaluation. Contact HQ AFOTEC/SAS for the latest metrics guidelines.

4.4. Evaluator Instructions. A handout is provided in attachment 1 to guide the evaluators during the evaluation. The following paragraphs provide additional detail, and should be understood by each evaluator prior to participating in a maintainability evaluation.

4.4.1. Question Interpretation. The evaluation team must determine how to apply each question by relying on their maintenance experience and sound judgment. Software maintainability is a measure of the ease with which a software product can be understood, modified, and tested. The evaluation team needs to keep this in mind when questions arise about the application of a particular question. The moderator is the final authority on question interpretation. Debate on issues should remain within the moderators interpretation of the question at hand.

4.4.2. Written Comments. In addition to the 1 to 6 responses illustrated above, it is important the evaluators include written comments so the moderator can substantiate the resulting scores. The evaluator comments are used by the moderator to formulate specific recommendations and highlight problem areas in the report.

4.5. Moderator Participation. In general, it is valuable for the moderator to participate as an evaluator (even if the scores are not counted) because the moderator can help guide discussions and record evaluator impressions and comments more effectively. This type of information is important when writing the evaluation report. If the moderator chooses not to be an evaluator, it is important that he/she pay close attention to the team discussions.

4.6. Analysis of Scores.

4.6.1. WINQAS should be used during the evaluation to record responses and analyze questionnaire response data. WINQAS calculates CSCI-level scores for each questionnaire (documentation, source code (module-level), source code (CSU-level), and implementation). These scores are computed using a weighted average of the characteristics applicable to each questionnaire. The default weightings are based on the number of questions within a characteristic (see table 6). For example, the descriptiveness characteristic in the module questionnaire accounts for 10 of 34 questions or 29.4 percent of the module-level questionnaire score. These WINQAS default weights can be modified for a specific program's needs.

Table 6. Number of Questions by Characteristic.

DOCUMENTATION QUESTIONNAIRE		SOURCE CODE MODULE QUESTIONNAIRE		SOURCE CODE CSU QUESTIONNAIRE		IMPLEMENTATION QUESTIONNAIRE	
Organization	4	Descriptiveness	10	Descriptiveness	2	Convention	6
Descriptiveness	21	Simplicity	11	Simplicity	1	Simplicity	7
Traceability	6	Modularity	4	Modularity	3	Modularity	7
General	1	Traceability	3	Traceability	2	Expandability	4
		Consistency	3	Consistency	2	General	1
		Testability	2	Expandability	1		
		Expandability	1	General	1		
		General	1				

4.6.2. The scores for each questionnaire should not be numerically combined for a total result. The aggregation strategy should be defined up-front in the software test concept. Results should be reported according to the test concept and guidelines in section 4.7.

4.7. Reporting. The maintainability evaluation report must capture the significant findings discovered during a maintainability evaluation. Since maintainability evaluations are done for different reasons (final OT&E reports, early feedback on systems, etc.), the resulting report should be tailored to meet the objectives of the evaluation. An evaluation conducted during dedicated OT&E might be more concerned with scores so that meet/does not meet standard criteria are reported. On the other hand, an evaluation performed during a software operational assessment (SOA) might be used to provide feedback to the developer for the purpose of improving the maintainability of the system. This report might concentrate on highlighting the problem areas and recommend an improvement strategy.

4.7.1. The characteristic scores for each category evaluation represent an important metric of maintainability information. The characteristics have been carefully chosen to represent universally important aspects of software maintainability. The scores at the characteristic level, along with the associated comments, represent a valuable source of information for the purpose of highlighting maintainability deficiencies.

4.7.2. When examining the numerical scores for individual questions, characteristics, and category evaluations, attention should be paid to the associated evaluator comments to try to substantiate the scores. For example, if the source code receives low scores on the questions dealing with embedded comments, the moderator (report writer) might note this in the report and back it up with specific evaluator comments noting that the source code was poorly commented. This approach to reporting (scores supported by evaluator comments) is an effective way to report maintainability evaluation results. Ideally, the scores should both reflect the evaluator's concerns and indicate the degree to which the software is deficient in each area (see table 7).

Table 7. AFOTEC Maintainability Thresholds.

SCORE RANGE	MAINTAINABILITY RATING
4.00 - 6.00	Good Maintainability
3.00 - 3.99	Marginal Maintainability
1.00 - 2.99	Unacceptable Maintainability

4.7.3. The moderator (report writer) has some flexibility in determining whether a system has met or failed to meet the standard (passed or failed). In the event that one or two of the evaluation categories (defined in figure 5) have scored below 3.0, the moderator should state that the system has not met the standard. However, in the event that all categories score above 3.0, the moderator should state that the system has at least marginally met standards. The range between 3.0 and 4.0 represents an area of possible score error induced by the natural variation in the population of potential evaluators (all evaluators are not the same). In general, the fewer evaluators the larger the score error (see section 6.6) on maintainability inspections).

4.7.4. In addition to stating whether the system has met/did not meet standards, the moderator should explain any serious problems in the report. The numerical scores themselves only tell part of the story. Evaluator comments and moderator observations can be very valuable in substantiating the numerical scores and helping the developer make improvements in the system and to the development process.

4.7.5. Figure 10 shows the sections that should be included in a final report. Example final reports can be obtained from HQ AFOTEC/SAS.

I. Executive Summary. Brief statement of the what was evaluated, who evaluated it and why, and the results at a high level (1/2 page).

II. Detailed Results:

1. Maintainability Evaluation Methodology. Brief description of software maintainability evaluation approach to evaluated maintainability. How was it used to evaluate this system (1 page).

2. Documentation Evaluation Results. Detailed results of the documentation evaluation. State specific findings here. Use comments provided by the evaluators in combination with the evaluation scores to provide as much information as possible about specific findings (1-3 pages).

3. Source Code Evaluation Results. Detailed results of the source code evaluation. Aggregate the results of the module and CSU-level evaluations to arrive at an overall rating for source code. Use comments provided by the evaluators in combination with the evaluation scores (1 page).

3.1 Module Evaluation Results. Detailed results of the module-level evaluation. State specific findings here. Use comments provided by the evaluators in combination with the evaluation scores to provide as much information as possible about specific findings (1-2 pages).

3.2 CSU Evaluation Results. Detailed results of the computer software unit (CSU) evaluation. State specific findings here. Use comments provided by the evaluators in combination with the evaluation scores to provide as much information as possible about specific findings (1-2 pages).

4. Implementation Evaluation Results. Detailed results of the implementation evaluation. State specific findings here. Use comments provided by the moderator in combination with the evaluation scores to provide as much information as possible about specific findings (1-3 pages).

III. Summary (or Recommendations). This section should be used as appropriate for the audience. This is a good place for recommendations on specific action to be taken to address the findings of the evaluations (1-2 pages).

Attachments
WINQAS Charts and Tables
List of Documents Used
List of Source Components Evaluated

Figure 10. Suggested Report Format.

5. Evaluator Guidelines. The following is a summary of the general philosophy that should guide each evaluator in answering each questionnaire.

5.1. Statements Not Questions. The evaluator will notice the "questions" are not questions at all. They are statements describing a desirable attribute of computer software documentation or coding practice. In "answering the question," the evaluator quantifies his or her subjective viewpoint as to what degree the desirable attribute is reflected in the system under evaluation. The average of all evaluator's answers on each question then provides the basis for an evaluation of the maintainability of the system.

5.2. Think "How maintainable would this system be for me?"

The primary consideration for the evaluator is
"How maintainable would this system be for me?"

The primary consideration for the evaluator is *"How maintainable would this system be for me?"* Keeping this thought in mind can simplify responses. There are situations/systems that are more/less maintainable because of their lack of some attributes described in the questions. Software maintainability is the measure of how easy it is to make changes to the software. You need to keep this in mind when questions arise about the application of a particular question to the software. Note that just because nothing could be done to make the software more maintainable does not mean that the software is inherently maintainable. Some algorithms are just going to be difficult to understand and maintain, and the scores should reflect this.

5.3. Share Information Not Answers. Each evaluator should share information he/she finds during the evaluation with other evaluators. However, specific answers (1 to 6) should not be shared so no one is swayed to answer a certain way. This ensures each evaluator's answers are truly his/her own opinion.

5.4. Examine Existence and Quality of Characteristics. Unless otherwise specified, the questions should be considered as investigating both existence and quality of a characteristic; that is, the evaluator is to first determine whether or not the attribute is present in the documentation, source listing, or implementation evaluation. If it is not present, the evaluator must decide if the absence enhances or detracts from software maintenance on the system. If it is determined that the attribute is present, the evaluator then tempers his/her answer to reflect the "goodness" in his/her judgment of that attribute in the system.

5.5. Not Evaluating Specification Compliance. You are evaluating the maintainability of the documentation, source listings, and implementation <u>as they stand</u>. Specification requirements or standards followed during development are irrelevant. If a question should be rated low, rate it that way <u>regardless</u> of whether or not it "meets spec".

5.6. Expertise in Evaluation Impacts Quality of the Air Force Product. Finally, bear in mind that you have been chosen for this evaluation because of your demonstrated expertise in software engineering and software support. That expertise and the professionalism you demonstrate in completing this evaluation will go far in providing the Air Force with a quality software product.

6. Issues/Lessons Learned. This section discusses different issues that may arise during the planning, execution, and reporting of a software maintainability evaluation. Please submit any new lessons learned to the OPR for incorporation into future releases of this publication.

6.1. What if ASETS can't be used? As ASETS matures, this problem will become less of a concern, but as of the date of this guide, several popular languages remain unsupported. If a manual sample is to be generated, care must be taken to ensure that the sample evaluated is representative of the population. Several approaches can be taken to meet this requirement. Partitioning by CSU purpose and/or size is one way to set up a sample. Once this partitioning is accomplished, a random sample could be taken from each group. Although not perfect, this method is better than a purely random sample. See section 4.2 on manual sample selection for more details. HQ AFOTEC/SAS can provide a sample in any case, regardless of whether ASETS can be used or not.

6.2. What if there is more than one language within a CSCI? When a single CSCI contains source code written in more than one language (e.g., JOVIAL and assembly language), a sample must be drawn from each language group. The distribution of source modules in the sample should approximate the distribution of the languages in the population. ASETS can be used to select a sample from each language type, or an ASETS sample can be mixed with a manual sample if necessary.

6.3. How do I aggregate the results to rate a multi-CSCI system?

6.3.1. When planning software OT&E on large software systems with multiple CSCIs, the STM must decide how to combine the results of each evaluation to arrive at an overall software supportability rating for the system. The aggregation strategy is a required part of the software test concept and must be determined before the system is evaluated. Failure to do so can jeopardize the objectivity of the results.

6.3.2. Remember that the ultimate objective is to rate the software system as "supportable" or "unsupportable." There may be several CSCIs in the system, each with its own supportability characteristics, but there can only be one system-level answer. In a multi-CSCI system, the first step is to rate the supportability of each CSCI using the results of all three evaluations (volume 2, volume 3, and volume 5). Then determine "weights" for the CSCIs according to their impact to system supportability. Finally, apply the weighted results to determine overall software supportability. The following paragraphs provide more detailed guidance for aggregating the results to answer the software supportability MOE or MOP.

6.3.3. First, rate each CSCI in the system for both product maintainability (using the results of the volume 3 evaluation or inspection) and adequacy of support processes/resources (combining the results of the volume 2 and volume 5 evaluations).

6.3.4. Next, develop a supportability rating for each CSCI in the system using the following rules of thumb (see table 8):

Table 8. CSCI Rating Strategy.

VOLUME 3 RESULTS	VOLUME 2/VOLUME 5 RESULTS	EXPLANATION	CSCI RATING
Good	Good	All's well!	Supportable
Bad	Good	Can good maintenance overcome a bad product?	Your call
Good	Bad	Can a good product overcome bad maintenance?	
Bad	Bad	Everything is bad.	Not Supportable

6.3.4.1. All is Well. The simplest case is when all evaluation results indicate the CSCI is supportable. If it looks, acts, and feels supportable.

6.3.4.2. Bad Product, Good Process. This is the next best case. If the results of the maintainability evaluation indicate the product is poor, but the results of the volume 2/volume 5 evaluations indicate the support processes and resources are good, then you must decide if the combination results in a supportable CSCI or not. Remember that a good process often results in continual improvement of the product; thus, the impact of poor product characteristics for maintainability may be temporary.

6.3.4.3. Good Product, Bad Process. If the results of the maintainability evaluation indicate the product is good, but the results of the volume 2/volume 5 evaluations indicate the support processes and resources are inadequate, then you must decide if a good product can overcome inadequate maintenance processes/resources. Remember that a poor process may result in continual erosion of the maintainability characteristics of the product; thus, the advantage of having a maintainable product initially may not last long.

6.3.4.4 Everything is Bad! Another simple case to analyze. Rate the CSCI as not supportable.

6.3.5. Next, aggregate the CSCI results to rate the entire system using the following rules of thumb (see figure 11):

6.3.5.1. All is Well (the Sequel). Again, when all CSCIs are rated as supportable, the answer is obvious. The system must be supportable as well.

6.3.5.2. Some CSCIs Are Supportable While Others Are Not. In this case, the test team must determine which CSCIs are most critical to the supportability of the system. If the critical ones are unsupportable, you should rate the system as unsupportable. Some factors in determining the "criticality" (or weighting) of each CSCI are:

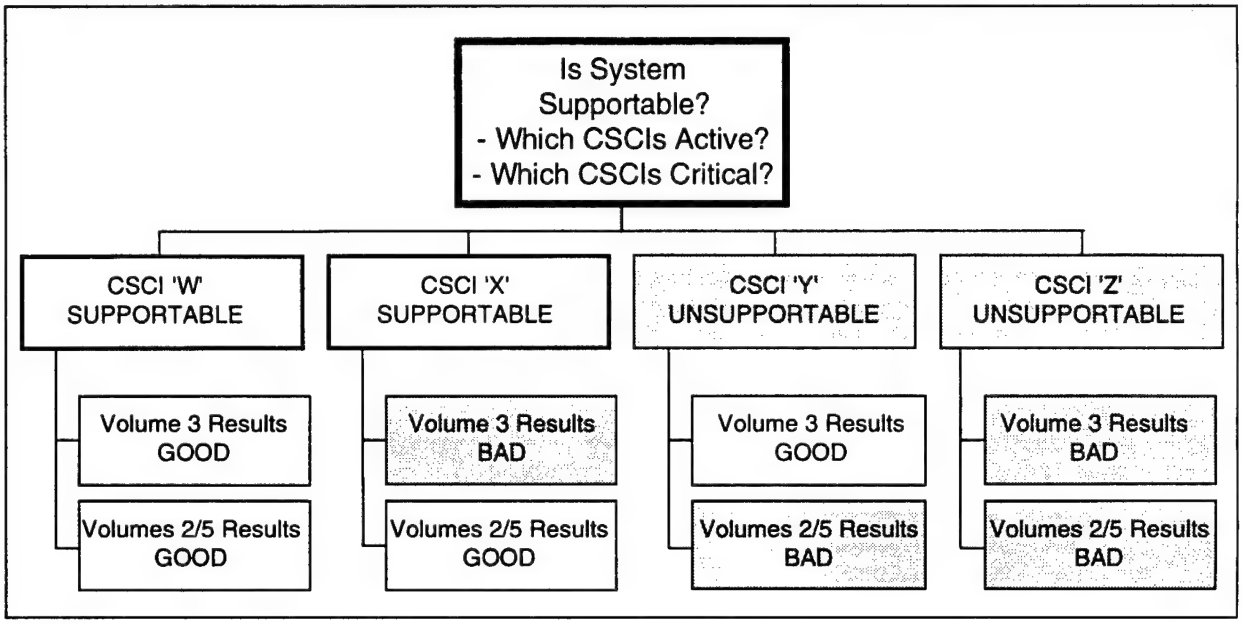

Figure 11. System Rating Strategy.

6.3.5.2.1. Expected Change Activity. CSCIs that are expected to undergo major changes after system deployment are more critical in terms of supportability then those that are expected to remain stable. Some criteria for determining expected change rate are:

6.3.5.2.1.1. CSCIs that are targeted for pre-planned product improvements should be given higher weight.

6.3.5.2.1.2. Often the "active" CSCIs for maintenance are identified in the CRLCMP or some other support planning document.

6.3.5.2.1.3. Alternatively, the results of the volume 6 maturity evaluation can help identify the CSCIs that are likely to continue to actively evolve after deployment.

6.3.5.2.2. Other Measures of Criticality. If any of the CSCIs are critical for any other reason (safety, cost, etc.), the test concept might require these CSCIs pass for the entire system to pass. The rationale here is that designing supportability into a system will also make post-deployment modifications more responsive and reliable.

6.3.5.3. Everything's Bad, Part II. If all CSCIs are rated as unsupportable, the system must be unsupportable as well.

6.4. How do I keep the evaluators focused and motivated? Always schedule comfortable facilities away from the evaluators' normal work areas. Since a lot of paper products are used during the evaluation, a large table and adequate seating are important. Also, ensure that restroom/snack facilities are nearby so evaluators don't waste excessive time on breaks or "wander off." Be sure to take plenty of breaks and try not to work more than 6 hours/day actually evaluating the system. Evaluators should be competent, non-disruptive, and willing to contribute

meaningfully to the evaluation. In the event that an evaluator becomes disruptive to the point that the evaluation cannot continue, that evaluator should be dismissed.

6.5. Who should I write the software maintainability evaluation report for? Since evaluations are at different times for different reasons, the evaluation report will not have a standard distribution. In some cases the report content should focus on whether the system passes or fails. In other cases the report should focus on providing feedback for the purpose of improving the system (Software Operational Assessment or other early evaluation). Most maintainability reports are signed/ approved by the test director or test manager, and you should discuss the focus of the report with them.

6.6. What if I can't get enough evaluators or a representative sample? When it is necessary to conduct an evaluation with fewer than five evaluators, or with a sample that is not representative of the system, then we call this a maintainability inspection. The fact that the minimum requirements for a normal maintainability evaluation have not been met should be documented in the evaluation report.

6.6.1. Impact of Number of Evaluators. When more evaluators are used in this evaluation, the resulting score is more accurate. In fact, using five evaluators, the evaluation score should fall within 0.55 of the true score. For example, an evaluation score of 4.5 using five evaluators means the true score lies somewhere between 3.95 and 5.05. The same score with ten evaluators is more precise (± 0.39), while an evaluation using only 2 evaluators yields a wider range (± 0.87). Table 9 shows the accuracy that can be expected based on the number of evaluators.[3]

Table 9. Confidence Bounds for Evaluation Score.

NUMBER OF EVALUATORS	ACCURACY
1	± 1.23
2	± 0.87
3	± 0.71
4	± 0.62
5	± 0.55
6	± 0.50
7	± 0.47
8	± 0.44
9	± 0.41
10	± 0.39

6.6.2. Impact of Non-Representative Sample. A sample is considered to be representative if it accurately reflects the maintainability characteristics of the entire software system under evaluation. If ASETS successfully processes all source code modules in the system, then the sample generated by ASETS is considered to be representative. When ASETS cannot be used or is not successful in processing some or all of the source code modules then a 10 percent sample (using the partitioning strategy described in section 3.2.2.3.1 is also considered to be representative. If these conditions can't be met, then the sample is not representative and the results of the maintainability evaluation cannot be automatically applied to the entire CSCI. The STM and DSE together must decide how to apply the results to answer the MOP.

6.7. How do I use the software maintainability evaluation on 4GL/CASE? First, it's important to realize that the software maintainability evaluation has been successfully applied to software developed in a fourth generation language (4GL) and/or using Computer Aided Software Engineering (CASE) tools. The following paragraphs outline some lessons learned from conducting maintainability evaluations on software with these features.

CASE tools simply automate some of the tasks involved in designing, programming, and/or testing software. Consider the impact of CASE tools only when they will be used to help maintain the software.

[3] BDM/TAC-78-698-TR, Analysis of Software Maintainability Evaluation Process, 6 December 1978

6.7.1. Computer Aided Software Engineering Tools. CASE tools simply automate some of the tasks involved in designing, programming, and/or testing software. Consider the impact of CASE tools only when they will be used to help maintain the software. In this situation, the moderator should (1) make sure all evaluators receive training on the CASE tools, and (2) ensure the tools are available during the evaluation to help answer questions. For example, if an on-line data dictionary is used, then the evaluators must know what its capabilities are and be able to query the dictionary during the evaluation to properly answer those questions that deal with the data dictionary.

6.7.2. Fourth Generation Languages. 4GLs are designed to improve the productivity achieved by high order languages and, often, to make computing power available to non-programmers. Features typically include an integrated database management system, query language, report generator, and screen definition facility. Software developed using a 4GL is often heavily dependent on CASE tools, and is difficult to evaluate outside the context of its CASE environment. The distinction between "documentation" and "source code" may not be clear, and the questionnaires contained in this evaluation guide may have to be tailored to account for this. HQ AFOTEC/SAS is actively investigating the impact of 4GL/CASE on software supportability, and the results will be incorporated into this volume when the research is completed.

6.7.3. 4GL Embedded in Higher Order Languages. Sometimes 4GL, such as Structured Query Language (SQL) statements, are embedded in source code components written in a more common high order language (e.g., Ada, FORTRAN, C). This embedded 4GL often causes problems with ASETS processing, and a sample may have to be selected manually. Otherwise, the evaluation proceeds normally by considering the impact of the embedded SQL statements when answering each maintainability questionnaire.

GEORGE B. HARRISON
Major General, USAF
Commander

SUMMARY OF EVALUATOR GUIDELINES

A1.1. General Guidelines.

A1.1.1. Work independently, share information—NOT ANSWERS.

A1.1.2. Complete all questionnaires specified by the evaluation moderator.

A1.1.3. Answer all questions. Use the response scale below (table A1-1). A response of 1 through 6 must be provided for each question (unless the question is to be scored a N/A).

Table A1-1. Question Response Scale

POINT VALUE	MEANING
6	Decidedly Agree
5	Moderately Agree
4	Somewhat Agree
3	Somewhat Disagree
2	Moderately Disagree
1	Decidedly Disagree
N/A	Not Applicable (*Don't Use Without Moderator Approval*)

A1.1.4. Please provide comments to substantiate your answers.

A1.2. Calibration Evaluations.

A1.2.1. The first time you answer each type of questionnaire, the moderator will stop the evaluation to review the scores and discuss any misunderstandings or disputes about how to interpret questions. All evaluators should have a uniform interpretation of how each question applies to the system, but you will not be forced to change any responses.

A1.2.2. The specific source modules and CSUs you will evaluate should be identified during the orientation briefing. All necessary materials will be provided in a work folder/handout.

A1.3. Assessment Procedures.

A1.3.1. The following sequence is generally used to complete questionnaires:

- Software Documentation Questionnaire.
- Source Listing Questionnaires.
- Update Documentation Questionnaire.
- Source Listing Questionnaires.

A1.3.2. Discuss "outlier" responses as directed by moderator.

A1.3.3. Complete remaining questionnaires in the sequence specified.

A1.3.4. Take care to complete all information on the answer sheets correctly.

A1.3.5. Review the explanation of the responses often and follow the guidelines. The primary consideration for the evaluator is *"How maintainable would this system be for me?"*

A1.3.6. Provide specific comments to substantiate question responses, especially high or low scores.

A1.3.7. Moderator completes implementation questionnaire before or after team evaluation of source code and documentation.

A1.4. Response Guidelines. (See table A1-2.)

Table A1-2. Question Response Scale with Explanations

RESPONSE	MEANING	EXPLANATION
6	Decidedly Agree	This response indicates the product being evaluated is excellent with respect to the attribute addressed.
5	Moderately Agree	This response indicates the product being evaluated is good with respect to the attribute addressed.
4	Somewhat Agree	This response indicates the product being evaluated is minimally acceptable with respect to the attribute addressed.
3	Somewhat Disagree	This response indicates the product being evaluated is unacceptable with respect to the attribute addressed.
2	Moderately Disagree	This response indicates the product being evaluated is poor with respect to the attribute addressed.
1	Decidedly Disagree	This response indicates the product being evaluated is terrible with respect to the attribute addressed.
N/A	Not Applicable	The statement does not apply to this software product. *Do not use without moderator approval.*

SOFTWARE DOCUMENTATION QUESTIONS

A2.1. The questions listed within this section are used to evaluate the overall format and content of the documentation for the computer program source code being evaluated. For the purposes of this evaluation, documentation includes information that describes the subject CSCI and is available to the future support organization. While this is commonly available in printed format, it may also be reviewed in other forms (such as on-line media).

A2.2. Document references in these questions are from MIL-STD-498. A table providing a mapping of MIL-STD-498 DIDs to DoD-STD-2167A and DoD-STD-7935A DIDs is provided in attachment 9.

A2.3. The characteristics evaluated in the documentation are Organization, Descriptiveness, and Traceability (see figure A2-1). Each statement in the questionnaire is used to evaluate an attribute of one of these characteristics.

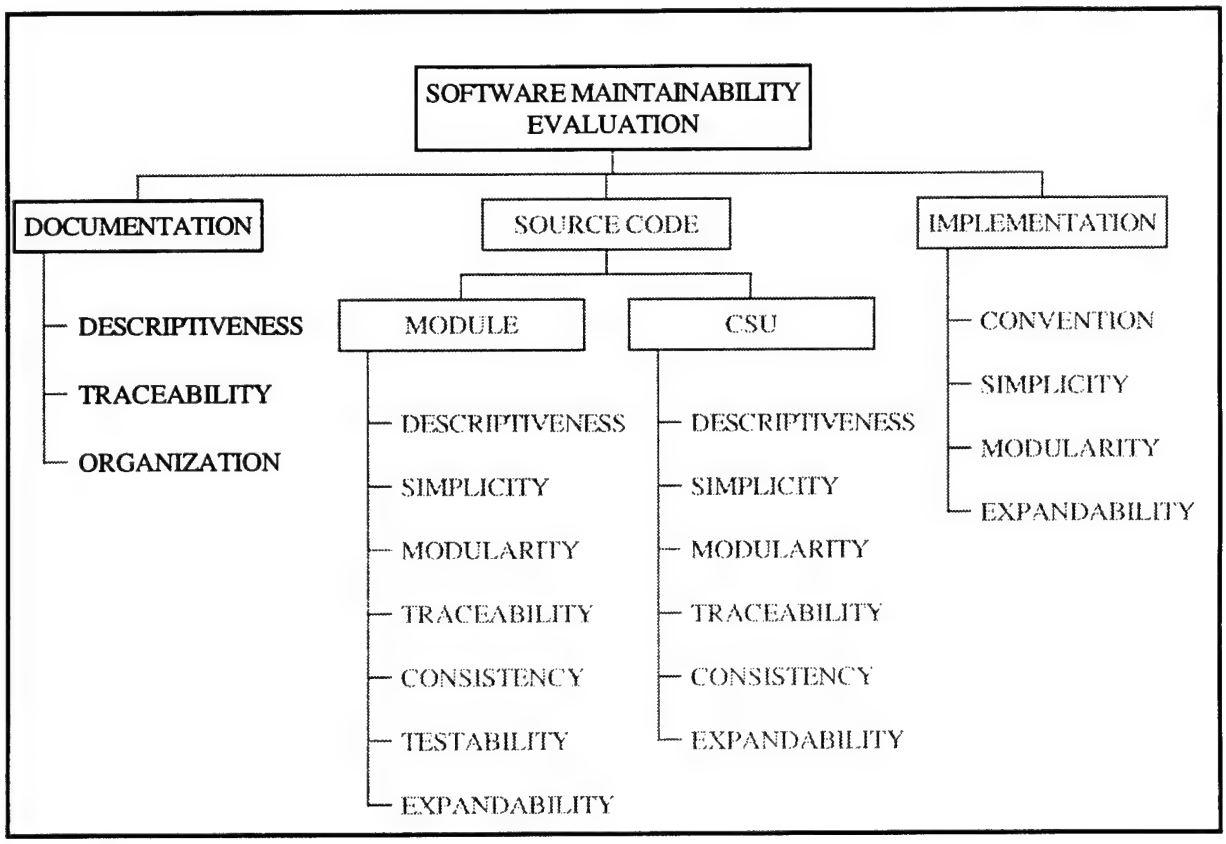

Figure A2-1. Documentation Maintainability Characteristics.

A2.4. Descriptiveness.

A2.4.1. Software documentation possesses the characteristic of descriptiveness when it contains useful information about requirements, assumptions, constraints, inputs/outputs, processing, components and their relationships, external interfaces, revision status, etc.

A2.4.2. The extent to which information is described is important to the software maintainer. This information is used as reference material to the maintainer, and is used to gain information about the system, aiding new personnel in understanding the system. If documentation lacks descriptiveness, the maintainer will have great difficulty in finding information about the system without an excessive amount of effort.

A2.5. Traceability.

A2.5.1. Software documentation possesses the characteristic of traceability when information can be traced between documents, and between documents and source listings.

A2.5.2. Documentation may be well written and well described, but still lack a clearly defined trail between top-level requirements and detailed implementation. The software maintainer must be able to trace any particular element from higher levels of program description down to executable code, and the reverse.

A2.6. Organization.

A2.6.1. Software documentation possesses the characteristic of organization when specific information can be easily located.

A2.6.2. The employment of standard conventions and easily understandable terminology greatly enhance the ease of use of any document. A hierarchical partitioning of the system's documentation from descriptions of less detail to descriptions of more detail should reflect the partitioning of the software.

D-1

STATEMENT: The documentation includes a useful version description document.

SCOPE: CSCI

CHARACTERISTIC: Descriptiveness

EXPLANATION: Some document should be readily available that describes the current operational version of each configuration-controlled program. At a minimum, a useful documentation master list must be included in the version description document. In hierarchical library systems, documents should be available describing each level of the library. For additional information about what information should be contained in the version description document reference MIL-STD-498 Data Item DI-IPSC-81442 (Software Version Description) or DoD-STD-2167A Data Item DI-MCCR-80013 (Version Description Document).

GLOSSARY: None

SPECIAL RESPONSE INSTRUCTIONS: Answer 1 if there is no version description document or equivalent.

EXAMPLE: None

D-2

STATEMENT: The terminology (acronyms, special terms) used in the documentation is understandable.

SCOPE: CSCI

CHARACTERISTIC: Descriptiveness

EXPLANATION: Most documents should include lists of acronyms and major terms which are standard throughout the project. A glossary of program-specific terms and acronyms would be useful in understanding the terminology used throughout the documentation.

GLOSSARY: None

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: None

D-3

STATEMENT: A program test plan is adequately described in the documentation.

SCOPE: CSCI and System Test Plans

CHARACTERISTIC: Descriptiveness

EXPLANATION: The test plan should include a methodology to test program performance at CSCI and system levels, test constraints, and a discussion of all program support tools needed to conduct the tests. This information is normally found in the software test plan (STP).

GLOSSARY:
TEST PLAN. A document prescribing the approach to be taken for intended testing activities. The plan typically identifies the items to be tested, the testing to be performed, test schedules, personnel requirements, reporting requirements, evaluation criteria, and any risks requiring contingency plans.

SPECIAL RESPONSE INSTRUCTIONS: Answer 1 if there is no useful test planning documentation.

EXAMPLE: None

D-4

STATEMENT: A useful set of test procedures and test data for CSCI or equivalent testing is contained in the documentation.

SCOPE: CSCI and system-level test procedures.

CHARACTERISTIC: Descriptiveness

EXPLANATION: Useful test procedures must provide adequate information to completely describe test inputs, outputs, and environment. One good test of the adequacy of the explanation of the program test procedures is for the evaluator to visualize how easy it would be to execute step-by-step one or more of the particular test procedures. If the information is not presented in a step-by-step fashion with a complete discussion of the test environment, test inputs, and expected test outputs, then the test will probably be difficult to perform. The overall description of the tests should be contained in the software test plan, while the detailed procedures should exist in the software test description (STD).

GLOSSARY:
COMPUTER SOFTWARE CONFIGURATION ITEM (CSCI). An aggregation of software that satisfies an end-use function is designated for configuration management and is treated as a single entity in the configuration management process.

SPECIAL RESPONSE INSTRUCTIONS: Answer 1 if there are no high-level test procedures.

EXAMPLE: Figure D-4 shows an excerpt of a CSCI test procedure.

OBJECTIVE: Create and maintain Dangerous Goods and DD Form 1387-2 templates that will be used to print the Dangerous Goods form and the DD Form 1387-2.

PRODUCTS: CU2A2280 (MNDANGER)

TEST SETUP/PREPARATION: N/A

PROCEDURE:
STEP 1. From the Air Freight Outbound menu, access the MNDANGER screen.
STEP 2. Create a Dangerous Goods template using the following information:

DANGEROUS/SIG TEMPLATE ID:	220
TEMPLATE TYPE: (DG/2):	DG
PROPER SHIPPING NAME:	BATTERY, WET, FILLED WITH ACID
UN CLASS/DIV NBR:	8
UN/NA ID NBR:	UN2794
PACKING GROUP:	III
HAZARD CLASS NAME:	CORROSIVE MATERIAL
HAZARD LABELS:	CORROSIVE
PAX/CARGO AIRCRAFT/CARGO AIRCRAFT ONLY:	P
RADIOACTIVE:	N (Default is N)
EMERGENCY CONTACT:	Use LIST function and select ALL OTHER HAZARDOUS MATERIALS.

STEP 3. Press F10 to save data.
STEP 4: Press F7 to enter query mode and type 220 in the DANGEROUS/SIG TEMPLATE ID field.
STEP 5. Press F8 to query the BATTERY record entered above.
STEP 6. Verify that the information retrieved matches the information previously entered.
STEP 20. Review the users manual to ensure it is clear and concise.
STEP 21. Verify that F2 help matches the specification.
STEP 22. Verify that ALT F2 help matches the specifications.

EXPECTED TEST RESULTS:
The expected test results are that:

STEPS 2-16. The user is able to create and maintain Dangerous Goods templates.
STEPS 17-18. The user is able to create DD Form 1387-2 templates.
STEP 19. Each field matches the specification/datamap.
STEP 20. The users manual is clear and concise.
STEP 21. F2 help matches the specifications.
STEP 22. ALT F2 help matches the specification.

Figure D-4. CSCI Test Procedures.

D-5

STATEMENT: A useful set of test procedures and test data for low levels of program testing is contained in the documentation.

SCOPE: Unit and integration test procedures.

CHARACTERISTIC: Descriptiveness

EXPLANATION: The test procedures should describe tests for the modules and CSUs of the program. These procedures are often used when a modification is made to a module or CSU and it is not feasible to retest the entire program. The overall description of the tests should be contained in the software test plan, while the detailed procedures may exist in the software development files.

GLOSSARY:
INTEGRATION TEST PROCEDURES. Set of test procedures designed to test a group of functionally related units.
UNIT TEST PROCEDURES. Set of test procedures designed to test an individual unit.

SPECIAL RESPONSE INSTRUCTIONS: Answer 1 if there are no low-level test procedures documented.

EXAMPLE: None

D-6

STATEMENT: Program initialization is adequately described.

SCOPE: CSCI

CHARACTERISTIC: Descriptiveness

EXPLANATION: The documentation should describe the user and system actions required to initialize the system. Every program does some initialization—even turnkey, embedded systems. Look for how well the analysis, design, and detailed design documents describe setting the initial program data and control states. The documentation should cover both the data and the steps required to initialize program operation within the system to resume processing after an interruption or begin processing after system shutdown start. This information may be found in the software installation plan (SIP), and in user and programmer manuals.

GLOSSARY:
INITIALIZATION. The preparatory steps required to set the initial program data and control states.

SPECIAL RESPONSE INSTRUCTIONS: Answer 1 if there is no description of program initialization.

EXAMPLE: None

D-7

STATEMENT: Program termination is adequately described.

SCOPE: CSCI

CHARACTERISTIC: Descriptiveness

EXPLANATION: The documentation should describe the user and system actions required to terminate the system. The documentation should cover both the data and the steps required to terminate program operation within the system. Both normal and abnormal termination of the program should be covered (when applicable). This information may be found in design description documentation, specifications, or in user and programmer manuals.

GLOSSARY:
TERMINATION. The steps required to set the final program data and control states (due to normal or abnormal termination).

SPECIAL RESPONSE INSTRUCTIONS: In some cases, such as missiles or bomb software, there is no need to perform any processing upon "termination." For these systems, answer N/A. Otherwise, answer 1 if there is no description of program termination.

EXAMPLE: For a few systems, the only program termination is turning off power. But most embedded applications still need to perform some type of processing before being turned off. Some examples include:

- Declassifying classified memory areas.
- Writing maintenance data out to a data transfer medium.
- Writing operations data (postflight debrief) information out to a data transfer medium.

D-8

STATEMENT: Each major system function is adequately described.

SCOPE: Functions within this CSCI.

CHARACTERISTIC: Descriptiveness

EXPLANATION: Personnel working on a specific function should have a complete description of that function. This information may be found in the software requirements specification (SRS) or in design description documentation. All applicable information for each function should be in one place within the documentation.

GLOSSARY: None

SPECIAL RESPONSE INSTRUCTIONS: Answer 1 if there is no description of each major function.

EXAMPLE: None

D-9

STATEMENT: Program-level control flow is adequately explained.

SCOPE: CSCI

CHARACTERISTIC: Descriptiveness

EXPLANATION: The documentation should include an overview section that describes the overall control flow in narrative or chart form. This question specifically addresses the explanation of the control flow, however it is organized. This information may be found in design description documentation.

GLOSSARY:
CONTROL FLOW. The sequence in which operations are performed during the execution of a computer program.
CONTROL FLOW DIAGRAM. A diagram that depicts the set of all possible sequences in which operations may be performed during the execution of a system or program. Types include box diagram, flowchart, input-process-output chart, state diagram.

SPECIAL RESPONSE INSTRUCTIONS: Answer 1 if there is no description (narrative or chart) of overall program control flow.

EXAMPLE: Figure D-9 illustrates a simplified flow of control between the interrupt handler, foreground processor, and background processor of the C-17 Flight Control Computer. Other design paradigms, such as object-oriented design, will have a different structure than that shown in this example, but program-level control flow organization should still be explained and documented.

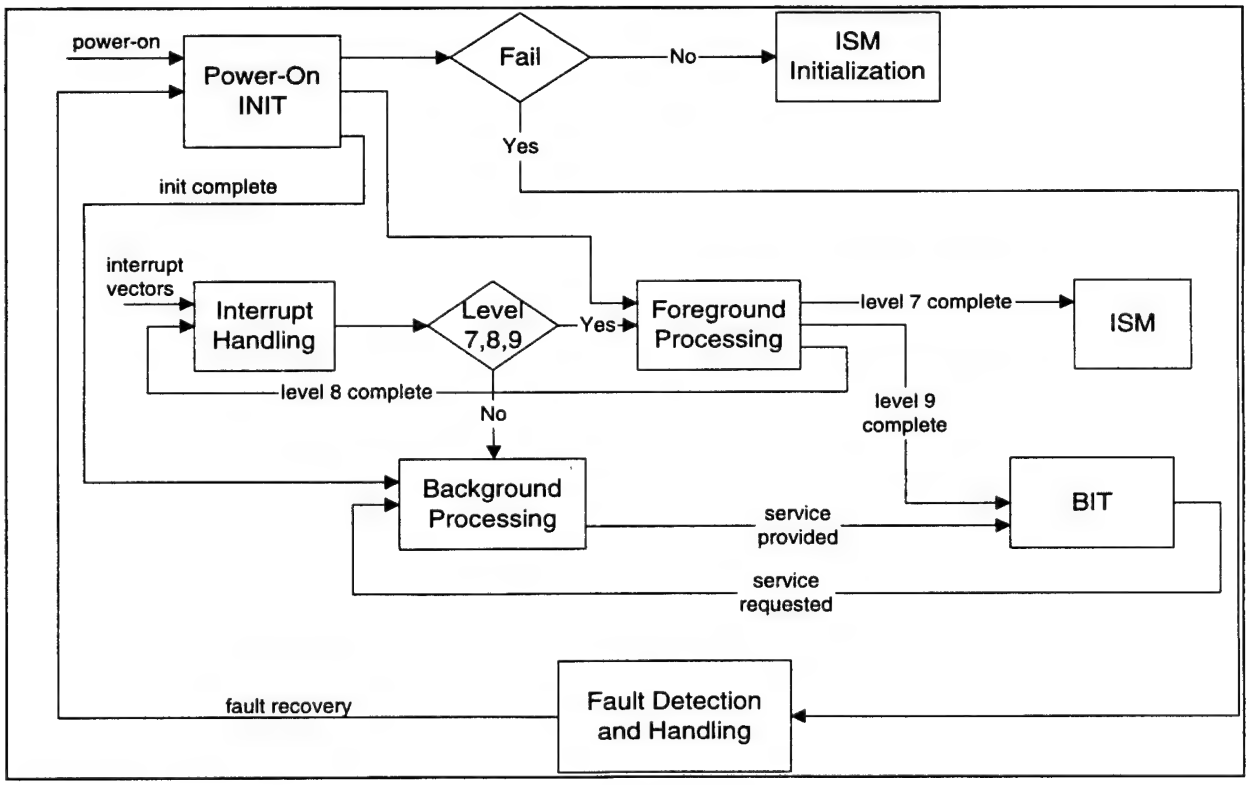

Figure D-9. Control Flow.

D-10

STATEMENT: Program-level data flow is adequately explained.

SCOPE: CSCI

CHARACTERISTIC: Descriptiveness

EXPLANATION: The documentation should include an overview section that describes the overall data flow in narrative or chart form. This question specifically addresses the explanation of the data flow, however it is organized. The documentation should facilitate easy tracing of data flow at all levels. This information may be found in design description documentation.

GLOSSARY:
DATA FLOW. The sequence in which data transfer, use and transformation are performed during the execution of a computer program.
DATA FLOW DIAGRAM. A diagram that depicts data sources, data sinks, data storage, and processes performed on data as nodes, and logical flow of data as links between the nodes.

SPECIAL RESPONSE INSTRUCTIONS: Answer 1 if there is no description (narrative or chart) of overall program data flow.

EXAMPLE: See figure D-10. Data flow and control flow diagrams may be integrated in the documentation.

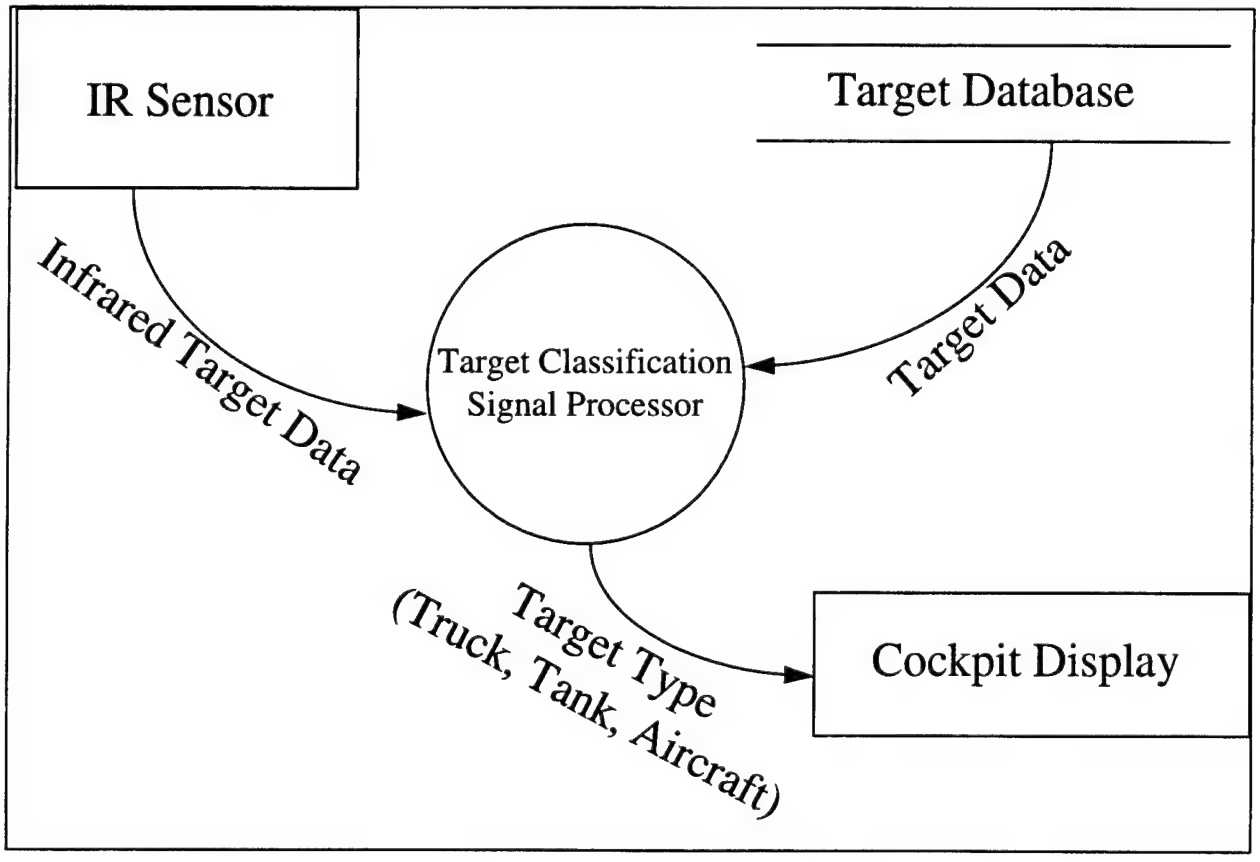

Figure D-10. CSCI-Level Data Flow.

D-11

STATEMENT: The data dictionary includes useful information (e.g., description, type, range, scaling, units, usage) about global data items, composite data items, and formal parameters passed as arguments between modules.

SCOPE: CSCI

CHARACTERISTIC: Descriptiveness

EXPLANATION: The documentation should contain a separate section in which all global and formal parameters are described to include information such as type, range, units, modules using, description, where declared, and any other pertinent information. This document should also describe the rationale behind any data partitioning that is done.

GLOSSARY:
COMPOSITE DATA TYPE. A data type, each of whose members are composed of multiple data items. For example, a data type called PAIRS whose members are ordered pairs (x,y).
DATA DICTIONARY. A collection of the names of all data items used in a software system, together with relevant attributes of those items.
FORMAL PARAMETER. A variable used in a software module to represent data or program elements that are to be passed to the module by a calling module.
GLOBAL DATA ITEM. Data that can be accessed by two or more nonnested modules of a computer program without being explicitly passed as parameters between the modules.

SPECIAL RESPONSE INSTRUCTIONS: Answer 1 if no data dictionary or its equivalent exists.

EXAMPLE: Table D-11 shows a sample data dictionary entry:

ITEM:	PRESSURE'ALT'SET
DESC:	The reference to calculate the current pressure altitude of the system
TYPE:	UNSIGNED INTEGER
SCAL:	Implicit division by 100 (1959 = 19.59)
UNIT:	Inches of Hg
USED:	GET'ALT, PRESSURE'ALT
SET:	CALIB'PRESSURE
DECL:	CALIBRATE.CMP
ATTR:	Position-independent, overlay, relocatable, global, shareable, readable, writeable

Figure D-11. Sample Data Dictionary

D-12

STATEMENT: The database design and interfaces are adequately described.

SCOPE: CSCI

CHARACTERISTIC: Descriptiveness

EXPLANATION: If the function of a database was just data storage, the organization would be simple. However, the database complexities arise from the fact that a database must show associations between data elements and the mechanisms to retrieve elements in a logical fashion. In order for maintainers to adequately maintain a database, the design and interfaces need to be well documented.

GLOSSARY:
DATABASE. A collection of interrelated data stored together in one or more computerized files.

SPECIAL RESPONSE INSTRUCTIONS: Answer N/A if no database exists.

EXAMPLE: Table D-12 shows an example data dictionary entry and a table that traces the data item to the modules that use them.

ORACLE Name	:	HHGS_TYPE_SHIPMENT_CODE
Data Dictionary Name	:	HOUSEHOLD_GOODS_TYPE_SHIPMENT_CODE
ID	:	DA200709
Size	:	1
Type	:	A
Unit of Measure	:	N/A
Range	:	S, N, D, BLANK
Accuracy	:	N/A
Precision	:	N/A
Description	:	IDENTIFIES HOUSEHOLD GOODS AS S FOR STORAGE IN TRANSIT; N FOR NONTEMPORARY STORAGE; OR D FOR DIRECT PROCUREMENT METHOD.

(TRACEABILITY OF ID)

DA ID	ORACLE NAME	DA LONG NAME	ORACLE TABLE	MODULES
DA200709	HHGS_TYPE_SHIPMENT_ CODE	HOUSEHOLD_GOODS_ TYPE_SHIPMENT_CODE	ACA_PR, PERS_PR	ACCOUNT_PROPERTY, PERSONAL_PROPERTY
ETC				

Figure D-12. Database Description

D-13

STATEMENT: The use of any complex mathematical model (technique, algorithm) is adequately explained.

SCOPE: CSCI

CHARACTERISTIC: Descriptiveness

EXPLANATION: The documentation should contain details on the use of any complex algorithm to include input requirements and limitations. If a published description or derivation is used, it should be referenced. Basic mathematics such as basic algebra functions (including trigonometric and geometric functions), equations, polynomials (including series), graphing of functions, basic manipulations, and Booleans don't need to be explained. The documentation should facilitate tracing the mathematics to the source code by giving the modules where the mathematics are found, what the variables in the documentation derivations are called in the source code, and so forth. This information may be found in design description documentation or in development plans.

GLOSSARY:
ALGORITHM. (1) A finite set of well-defined rules for the solution of a problem in a finite number of steps; for example, a complete specification of the sequence of arithmetic operations for evaluating sin x to a given precision. (2) Any sequence of operations for performing a specific task.
COMPLEX MATHEMATICAL MODEL. Examples of complex mathematical models are Fourier transform, Laplace transform, numerical integration/differentiation, control theory, statistical techniques.

SPECIAL RESPONSE INSTRUCTIONS: Answer N/A if there are no complex mathematical models.

EXAMPLE: None

D-14

STATEMENT: The documentation adequately describes the external interfaces.

SCOPE: This question concerns interfaces between this CSCI and other CSCIs, HWCIs, or the external environment.

CHARACTERISTIC: Descriptiveness

EXPLANATION: The documentation should describe what data are input/output, the form of the data, any limitations on the data, sources/destinations of data, how they are input/output, their formats, and how they are processed—what happens to the inputs when they are received, and what is done to the outputs before they are handed off. Any explicit action the programmer must perform or convention the programmer must follow to accomplish I/O should be explained as well. This information is normally included in the Software Requirements Specification (SRS), Interface Requirements Specification (IRS) System/Subsystem Specification (SSS), Interface Design Description (IDD) or other design documents and should include graphical data flows as well as text descriptions.

GLOSSARY:
EXTERNAL INTERFACE. A boundary at which independent computer programs interact or at which a computer program interacts with hardware (e.g., program input/output data or interrupts) or users (e.g., human-computer interface).

SPECIAL RESPONSE INSTRUCTIONS: Answer 1 if no explanation of external interfaces and their processing exists.

EXAMPLE: None

D-15

STATEMENT: The timing scheme (to include the use of concurrency) for the program is adequately described.

SCOPE: CSCI

CHARACTERISTIC: Descriptiveness

EXPLANATION: This question is focused on real-time systems rather than throughput or response time for nonembedded systems.

The program documentation should include a description of the overall timing requirements and the timing scheme designed to satisfy process scheduling. This description should include the allocation time for each major system function and the timing relationships among major functions.

For some applications, the timing scheme for a program is not important. But there are many applications that have maximum response times allowed, and this should be explained.

For any real-time system, there must be some timing requirements that must be explained. They should include a description of the real-time scheduler (the process that monitors and coordinates running processes to ensure timing requirements are met). Additionally, concurrency (more than one process running at the same) may be used to implement the software so that it meets timing requirements. This information may be found in a timing and sizing report or in requirement specifications and design description documents.

GLOSSARY:

CONCURRENT PROCESSES. Processes that may execute in parallel on multiple processors or asynchronously on a single processor. Concurrent processes may interact with each other, and one process may suspend execution pending receipt of information from another process or the occurrences of an external event.

REAL-TIME SYSTEM. Software that measures, analyzes, or controls real-world events as they occur. A real-time system must respond within strict time constraints.

TIMING SCHEME. The implementation of the timing requirements with consideration to time slicing, time sharing, priority levels, or rate groups as applied to the overall sequencing and execution of program functions.

SPECIAL RESPONSE INSTRUCTIONS: Answer N/A if timing is not a factor and concurrency is not used or the software is not part of a real-time system.

EXAMPLE: None

D-16

STATEMENT: Any use of recursive or reentrant programming is adequately described in the documentation.

CHARACTERISTIC: Descriptiveness

EXPLANATIONS: The documentation should specify within a general "program design considerations" section or the individual module description section whether recursion or reentrancy is to be used. Many languages (or at least a particular implementation of a compiler) do not allow recursive or reentrancy code. Some languages (e.g., stack-oriented languages like JOVIAL or Ada) allow recursion as a natural language capability.

The evaluator should get an overall impression of how much recursion and reentrant programming is a part of the overall program design. If done with care, some use of recursion and/or reentrancy can simplify the overall program design even though the particular modules which are recursive and/or reentrant will probably be harder to maintain because of those concerns.

Any use of recursion or reentrancy should be described in detail, clearly showing the advantages of and concepts behind this use. This information is normally part of the module-level detailed design.

GLOSSARY:
RECURSIVE PROGRAMMING. A process in which a software module calls itself.
REENTRANT PROGRAMMING. Pertaining to a software module that can be entered as part of one process while also in execution as part of another process and still achieve the desired results.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: None

D-17

STATEMENT: Recovery from error conditions is adequately described.

SCOPE: Errors internally or externally generated to the CSCI.

CHARACTERISTIC: Descriptiveness

EXPLANATION: The documentation should include an explanation of overall error processing. This description should include a description of the recovery of the program from error conditions generated external to the program but affecting its capability to function. This explanation will usually concern the recovery from invalid data. This information may be found in design description documentation, and in system and requirement specifications.

GLOSSARY:
RECOVERY. The restoration of a system, program, database, or other system resource to a state in which it can perform required functions.

SPECIAL RESPONSE INSTRUCTIONS: Answer 1 if no description of recovery from externally or internally generated error conditions exists.

EXAMPLE: Recovery from external power interrupt, recovery from illegal data.

D-18

STATEMENT: Any dynamic allocation of resources is adequately explained.

SCOPE: CSCI

CHARACTERISTIC: Descriptiveness

EXPLANATION: The documentation describing the functional/detailed program specifications should include a section that explains how dynamic allocation is implemented by the program. Software design description and system/subsystem design description documents may contain this information.

GLOSSARY:
DYNAMIC ALLOCATION. A computer resource allocation technique in which the resources assigned to a program vary during program execution, based on current need.

SPECIAL RESPONSE INSTRUCTIONS: Answer N/A if it is clear that dynamic allocation of resources is handled only by the operating system, task scheduler, or equivalent or if no dynamic allocation of resources exists.

EXAMPLE: The priority scheme or timing allocation for particular "rate" groups may depend upon certain phases of a mission and dynamically change on that basis. Likewise, assignment of control of tape drives, disks, communication lines, or other hardware may be done in some dynamic manner.

D-19

STATEMENT: Storage requirements for this CSCI are adequately described in the documentation.

SCOPE: CSCI

CHARACTERISTIC: Descriptiveness

EXPLANATION: Allocated storage requirements for each major function should be described in the documentation. Even if a program does not have any "critical" storage requirements, there should be an explanation in the documentation covering the program's environment. These "storage requirements" specifically address the amount of program memory, but other forms of memory such as disk space should also be considered. In a virtual memory environment, topics such as page size, page replacement algorithms, and roll-in/roll-out mechanisms should be described. This information may be found in design description documentation, specifications, or in user and programmer manuals.

GLOSSARY: None

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Figure D-19 is an excerpt of the memory allocation map and program memory allocation table from the C-17 Flight Control Program.

"The FCC CPCI is allocated across three 1750A processors. The hardware memory map depicting data and instruction storage are contained in Tables ..."

Table XXIII Data Memory Map

ADDRESS (HEX)	PROCESSOR		
	IOP	CPA	CP3
0000-003F	Interrupt Vectors	Interrupt Vectors	Interrupt Vectors
0040-007F	Double-Mapped PMEN	Double-Mapped PMEN	Double-Mapped PMEN
• • •	• • •	• • •	• • •
6000-DFFF	Program Instruction	Nonoverlay Program Instruction	Nonoverlay Program Instruction

Table A2-4. XXV FCC CPCI Program Memory Allocation.

CPU	EXE	ISM	OSM	CLC	DM	BIT	TOTAL	AVAILABLE	%USED
IOP	11322	2002	141	128	18086	25674	57353	57344	100.0
CPA	4689	6790	4105	36531	1569	4947	58631	90112	65.0
CPB	4635	6152	4715	18201	1625	26799	62126	90112	68.9
total	20646	14944	8060	54860	2120	57420	178810	237568	74.9

Figure D-19. Memory Allocation Map.

D-20

STATEMENT: The documentation includes a description of how to alter basic data storage sizes and includes a memory map that illustrates how address space is allocated to each function.

SCOPE: CSCI

CHARACTERISTIC: Descriptiveness

EXPLANATION: How to alter the capacity of data storage is not always obvious. Often, storage has been carefully allocated to interface with various portions of the program. Documentation narrative should not only describe how to alter basic data storage sizes, but should also identify those interfaces which might be impacted by such changes.

Changing the definition of a data structure will impact the modules that use it. Therefore, the documentation should contain a list of "affected modules" for each data structure so changes to the structure can be accompanied by appropriate changes to the modules. Automated tools exist for tracking down affected modules, and should be used in preference to manually updating lists.

How and where the functions within a program are allocated to processor memory must be known to enable program modifications. This is especially important in embedded systems, where extra memory space is usually at a premium. A chart or memory map should show how the address space is allocated to each function. The amount and location of spare memory should also be described, along with any shared memory locations. This information may be found in design description documentation, specifications, or in programmer manuals.

GLOSSARY:
MEMORY MAP. A diagram that shows where programs and data are stored in a computer's memory.

SPECIAL RESPONSE INSTRUCTIONS: Answer N/A if memory allocation is handled by the operating system or equivalent system scheduler and there is no use of shared memory.

EXAMPLE: Figure D-20 shows a sample system memory map:

```
Location        Contents
FFFFh   <------
 ||             Reserved for future expansion
D000h   <------
CFFFh   <------
 ||             System data base
C000h           <------
BFFFh   <------
 ||             Unused
B000h           <------
AFFFh   <------
 ||             Reserved
A400h   <------
A3FFh   <------
 ||             R/W Memory
6000h           <------
5FFFh           <------
 ||             EEPROM's constant segment
4000h           <------
3FFFh           <------
 ||             EEPROM's code segment
0000h           <------
```

Figure D-20. System Memory Map

D-21

STATEMENT: Each CSU/module purpose and processing are adequately described in the documentation.

SCOPE: Examine the CSU/module descriptions in the documentation.

CHARACTERISTIC: Descriptiveness

EXPLANATION: Each CSU/module should have one distinct purpose, and that purpose should be described. This description will be a concise explanation of the CSUs overall processing.

Different types of errors possible and their effects should be discussed along with any other asynchronous events. This information may be found in software development files (SDF), design description documentation, and specifications.

GLOSSARY:
COMPUTER SOFTWARE UNIT (CSU). An element specified in the design of a CSCI that is separately testable and contains a group of modules that are (usually) functionally related. As a rule of thumb, a CSU is the level at which a SDF is maintained.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: None

D-22

STATEMENT: Two-way traceability exists from the high-level requirements to the low-level design implementation.

SCOPE: CSCI

CHARACTERISTIC: Traceability

EXPLANATION: There must be a clear correlation first between stated system requirements and high-level functions, and then from high-level functions down to low-level design implementation. This question should be viewed as examining a maintainer's ability to trace between top-level (system) requirements through intermediate-level (function) descriptions and finally to implementation within the source code. The developer should have some requirements tracing description, either on-line or through a requirements matrix. The developer should have some capability for on-line requirements traceability.

GLOSSARY: None

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Table D-22 shows a matrix that shows a direct correspondence between "requirements" and "function" documentation. Or a table may show where each requirement is implemented through successive levels of detail:

SYSTEM REQUIREMENTS	SRS	SDD
Paragraph 7.2.6.1 EW Threat Modeling	D-503-11652 Paragraphs 3.4.1-3.4.10	D-1503-11652-1 Paragraphs 3.8.1-3.8.6, 4.2.1-4.2.9
Paragraph 7.2.6.2 EW Threat Capacity	D-503-9876 Paragraphs 1.7.1-1.7.6	D-1503-9876-1 Paragraphs 3.2.7-3.2.9

Figure D-22. Requirements Traceability.

D-23

STATEMENT: It is easy to trace the program control flow at all system levels.

SCOPE: CSCI

CHARACTERISTIC: Traceability

EXPLANATION: When making changes to a module, the maintainer must know all of the called and calling modules to predict impacts. Therefore, some form of functionally oriented presentation of each of the program components should be available in the documentation. This could take the form of structure charts, flowcharts, etc. The documentation should facilitate easy tracing of control flow at all program levels. Question D-9 addresses the descriptiveness of the control flow information. This question looks only at the control flow. Data flow is handled in the next question.

GLOSSARY:
CONTROL FLOW. The sequence in which operations are performed during the execution of a computer program.
CONTROL FLOW DIAGRAM. A diagram that depicts the set of all possible sequences in which operations may be performed during the execution of a system or program. Types include box diagram, flowchart, input-process-output chart, state diagram.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Figure D-23 shows program control flow.

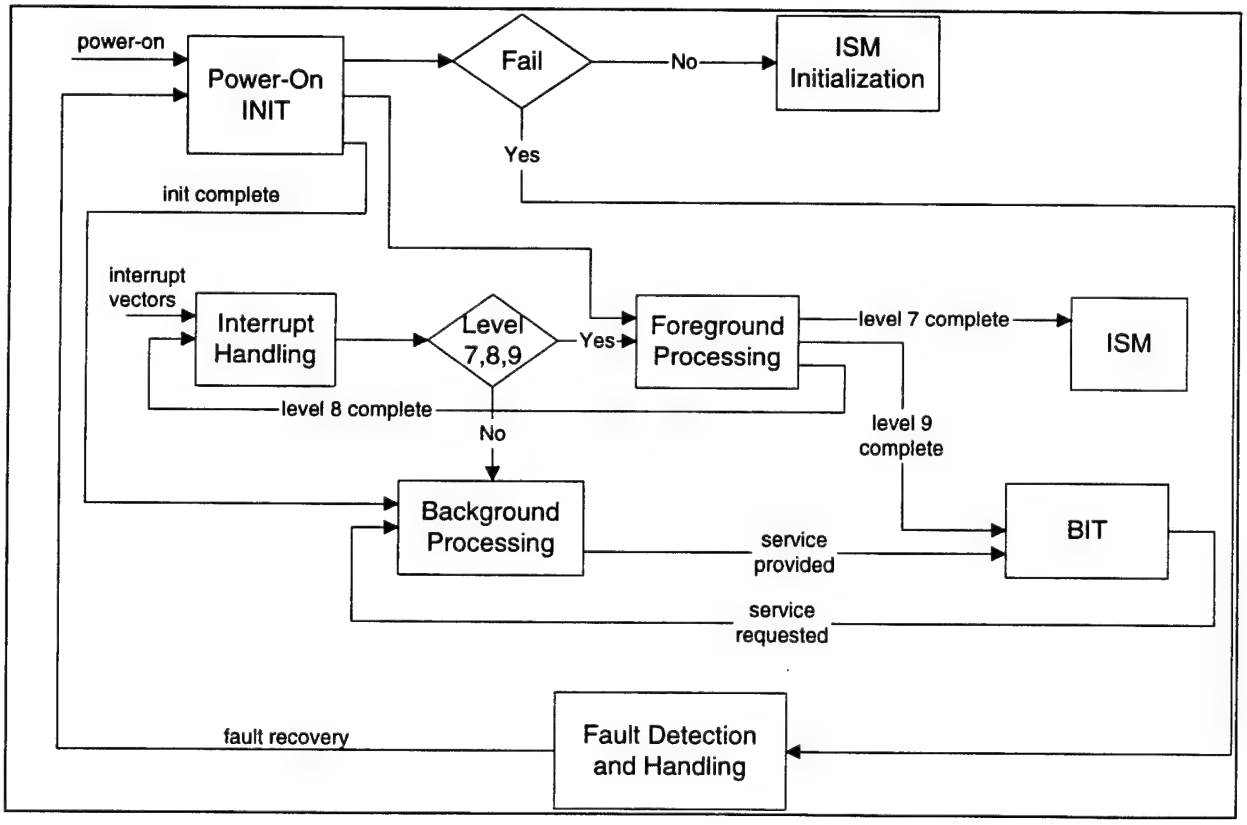

Figure D-23. Program Control Flow.

D-24

STATEMENT: It is easy to trace the data flow of the program at all system levels.

SCOPE: CSCI

CHARACTERISTIC: Traceability

EXPLANATION: When making changes to a module, the maintainer must know all the program data interfaces to anticipate impacts. Therefore, some sort of data flow diagrams or graphical representations of the data passed between modules in a program is necessary. Whatever method is used to present the flow, the presentation should be understandable and complete.

GLOSSARY:
DATA FLOW. The sequence in which data transfer, use and transformation are performed during the execution of a computer program.
DATA FLOW DIAGRAM. A diagram that depicts data sources, data sinks, data storage, and processes performed on data as nodes, and logical flow of data as links between the nodes.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Figure D-24 illustrates the data flow between processes in the C-17 Flight Control Computer.

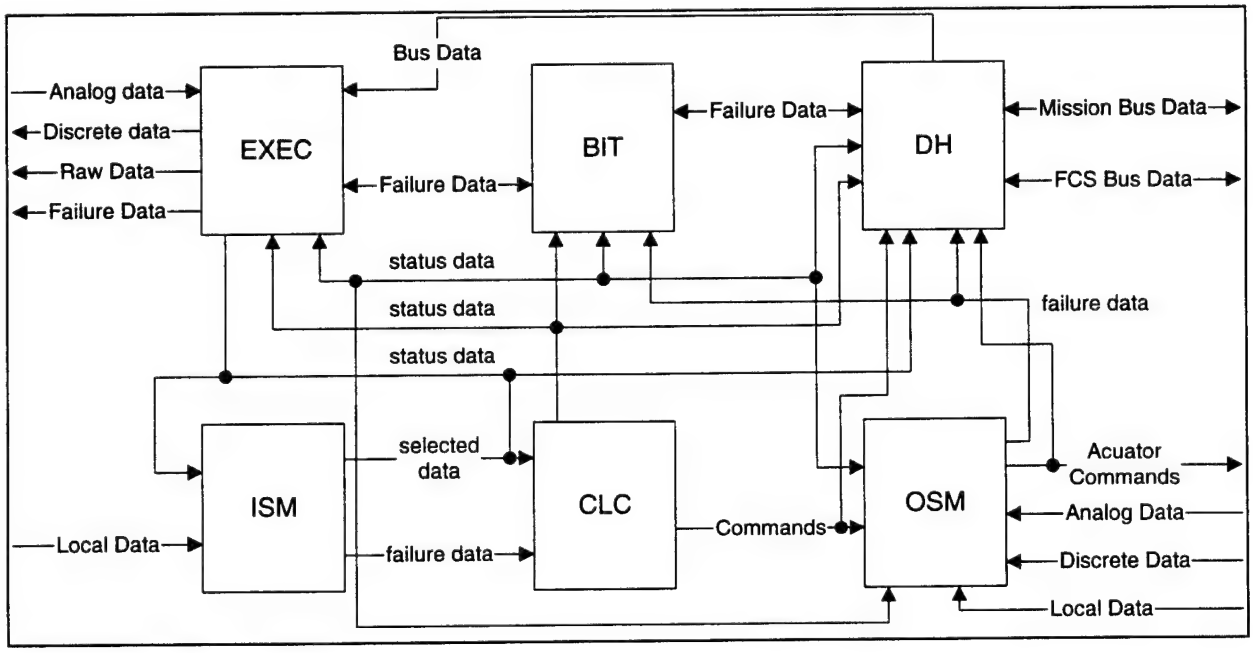

Figure D-24. Flight Control Computer Data Flow.

D-25

STATEMENT: Global data items and data structures are easily traceable to the modules which use them.

SCOPE: CSCI

CHARACTERISTIC: Traceability

EXPLANATION: The maintainer must be able to determine which modules are affected if changes are made to a global data element or data structure. Information should be available for any data with a scope greater than a single module.

GLOSSARY:
GLOBAL DATA ITEM. Data that can be accessed by two or more nonnested modules of a computer program without being explicitly passed as parameters between the modules.

SPECIAL RESPONSE INSTRUCTIONS: Answer 6 if there are no global data.

EXAMPLE: None

D-26

STATEMENT: Composite data type descriptions are easily traceable to the modules which use them.

SCOPE: CSCI

CHARACTERISTIC: Traceability

EXPLANATION: None

GLOSSARY:
COMPOSITE DATA TYPE. A data type, each of whose members are composed of multiple data items. For example, a data type called PAIRS whose members are ordered pairs (x,y).

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Figure D-26 is an example of a record in Ada.

```
Type File_Description_Type IS
    RECORD
        Name_Part       : String (1..8);
        Extension_Part  : String (1..3);
        Length_Part     : Natural;
        Location_Part   : Integer Range 0..511;
    END RECORD;
```

Figure D-26. Ada Record Declaration.

D-27

STATEMENT: Database items are easily traceable to the modules which use them.

SCOPE: CSCI

CHARACTERISTIC: Traceability

EXPLANATION: The data used in a database do not change in the same way as data items in executable source code. The maintainer must be able to determine which modules are affected if changes are made to a database element or data structure.

GLOSSARY:
DATABASE. A collection of interrelated data stored together in one or more computerized files.

SPECIAL RESPONSE INSTRUCTIONS: Answer N/A if no database exists.

EXAMPLE: Table D-27 shows an example data dictionary entry and a table that traces the data item to the modules that use them.

ORACLE Name	: HHGS_TYPE_SHIPMENT_CODE
Data Dictionary Name	: HOUSEHOLD_GOODS_TYPE_SHIPMENT_CODE
ID	: DA200709
Size	: 1
Type	: A
Unit of Measure	: N/A
Range	: S, N, D, BLANK
Accuracy	: N/A
Precision	: N/A
Description	: IDENTIFIES HOUSEHOLD GOODS AS S FOR STORAGE IN TRANSIT; N FOR NONTEMPORARY STORAGE; OR D FOR DIRECT PROCUREMENT METHOD.

(TRACEABILITY OF ID)

DA ID	ORACLE NAME	DA LONG NAME	ORACLE TABLE	MODULES
DA200709	HHGS_TYPE_SHIPMENT_CODE	HOUSEHOLD_GOODS_TYPE_SHIPMENT_CODE	ACA_PR, PERS_PR	ACCOUNT_PROPERTY, PERSONAL_PROPERTY
ETC				

Figure D-27. Database Traceability

D-28

STATEMENT: Each physically separate part of the documentation includes a useful table of contents and index.

CHARACTERISTIC: Organization

EXPLANATIONS: Each separately bound part of the set of documentation for this program has its own table of contents to help locate program information. Considerations for a good table of contents include, but are not limited to, the amount of subparagraphing, use of indentation, and the use of descriptive titles.

A useful index does not necessarily include every word in a document. Instead, it should have the important concepts for the specific document listed. Very few software documents include an index. But just because one is not included does not mean that it would not be a very useful list to have. They are typically very easy to generate, so no document should be without one. If the documentation is available on-line, then this question can be answered based on how difficult it is to access a document and search for key words or phrases.

GLOSSARY: None

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLES: None

D-29

STATEMENT: A useful set of standards has been followed for the organization and content for the documentation of each major part of the program.

SCOPE: CSCI

CHARACTERISTIC: Organization

EXPLANATION: The documentation should be examined for consistent format in the presentation of graphic and narrative content. Consistent documentation means that the maintainer can spend less time learning the organization of the documentation and more time learning the content. The documentation should be skimmed for adherence to standards, including standards for both the narrative and graphical portions. There should be a separate part of the documentation for each major functional part of a program, and these parts should have the same documentation format. The VDD is another example of what should be a separate document

GLOSSARY:
STANDARDS. Mandatory requirements employed and enforced to prescribe a disciplined uniform approach to software development, that is, mandatory conventions and practices are, in fact, standards.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: None

D-30

STATEMENT: The documentation is organized as a useful, systematic description of the program from levels of less detail to levels of more detail.

SCOPE: CSCI

CHARACTERISTIC: Organization

EXPLANATION: The documentation produced during a software development effort should successively describe requirements definition, preliminary design, detailed design, operation/maintenance, testing, etc. This will reflect a natural progression of program description from levels of less detail to levels of more detail.

GLOSSARY: None

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Figure D-30 shows a possible set of software development documentation.

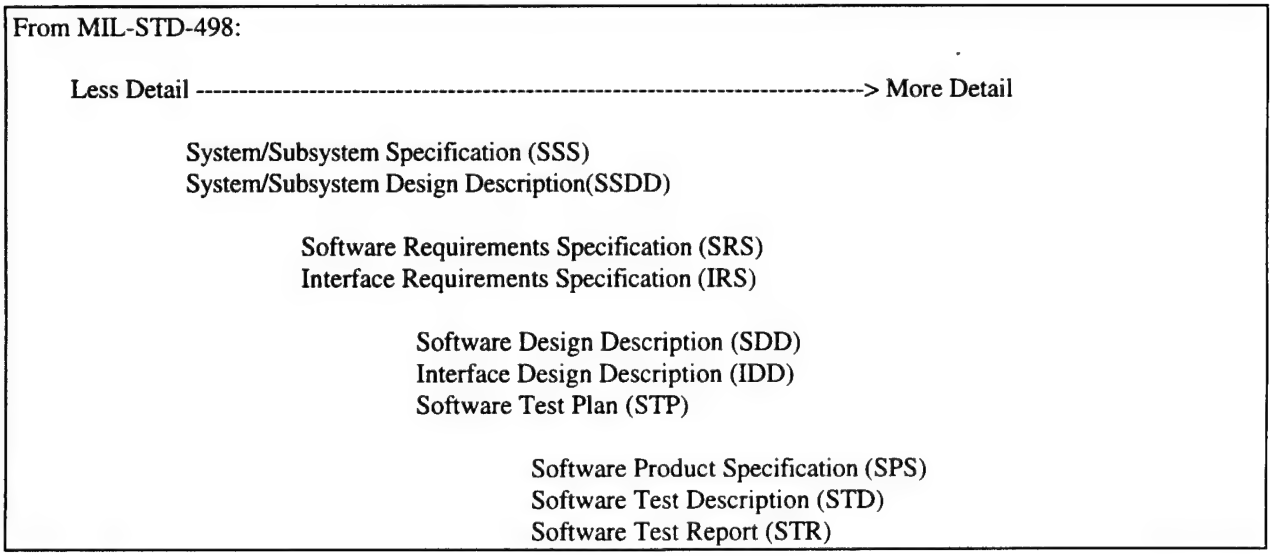

Figure D-30. Software Development Documentation.

D-31

STATEMENT: It is easy to locate specific information within the documentation.

CHARACTERISTIC: Organization

SCOPE: CSCI

EXPLANATION: The evaluator should conceptualize the need for locating a specific piece of information, and then check the documentation for the effort required to locate the information.

Sampling major parts of the documentation for the amount of cross-referencing and the essential nature of the cross-referencing is one method of obtaining an impression as to level of agreement/disagreement.

Examine the documentation for useful tables of contents and indices, organization of the documentation along functional lines, and separation of the documentation by purpose (specification, functional description, interface descriptions, etc.).

GLOSSARY: None

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Examples of specific information to be easily located are: information about a specific module (parameter lists, calling modules, called modules, etc.), external interface information, and global data.

D-32

STATEMENT: Overall, it appears that characteristics of the program documentation contributes to the maintainability of the system.

SCOPE: CSCI

CHARACTERISTIC: General

EXPLANATION: Use this question to rate the overall maintainability of the documentation. In addition to a 1 through 6 answer, provide any comments on any maintainability aspects of the documentation not covered by the previous questions.

GLOSSARY: None

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: None

MODULE SOURCE LISTING QUESTIONS

The questions in this section are used to evaluate the overall format and content of the source code module and to evaluate the consistency between the module's documentation and source listing.

The module characteristics evaluated are modularity, descriptiveness, consistency, simplicity, expandability, traceability, and testability (see figure A3-1). Each question in the questionnaire is used to evaluate an aspect of one of these module source listing characteristics.

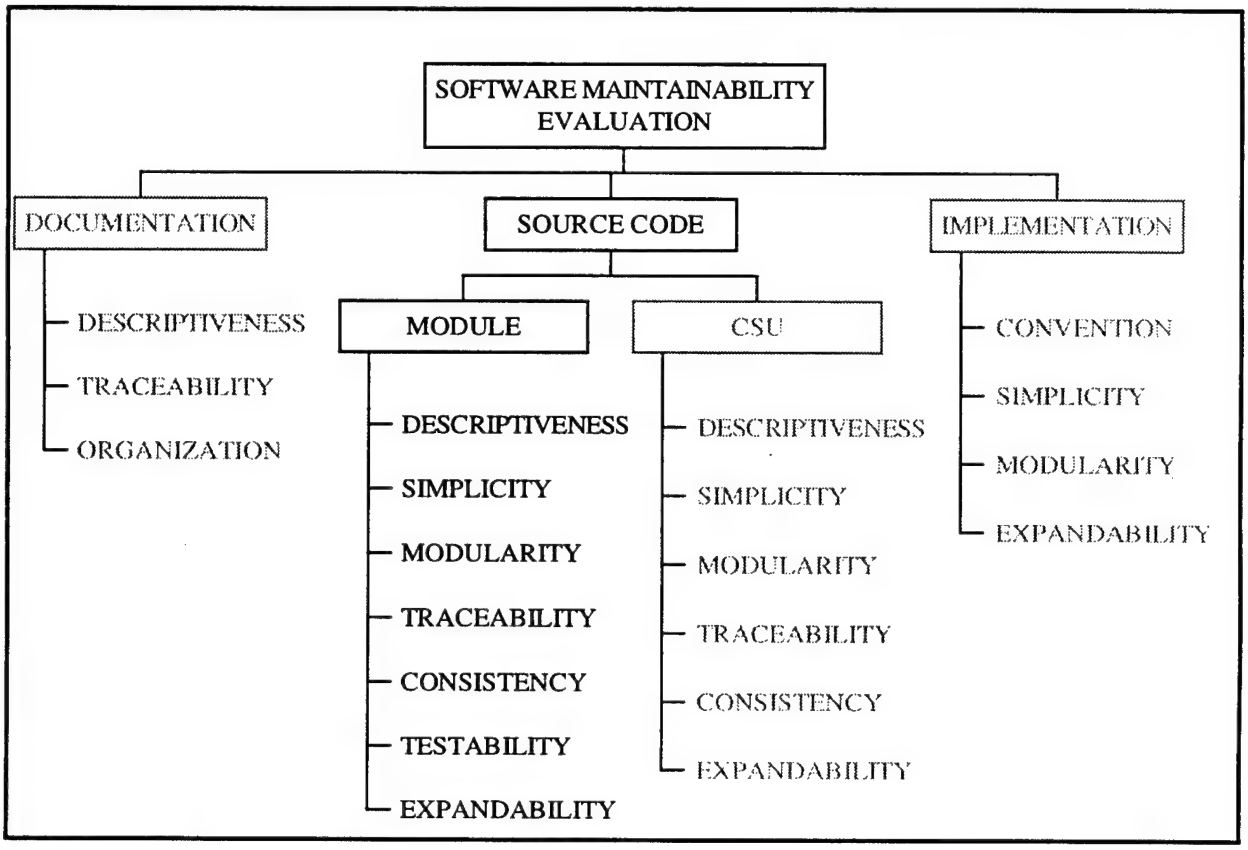

Figure A3-1. Module (Source Code) Characteristics.

A3.1. Descriptiveness.

A3.1.1. Modules are descriptive when they contain information about:

- objectives
- assumptions
- constraints
- inputs
- processing
- outputs
- components
- relationships to other code units
- and revision status

A3.1.2. Meaningful identifiers, indentation, and embedded comments are a few examples of descriptiveness items that aid the maintainer in understanding source code.

A3.2. Simplicity.

A3.2.1. Simple modules are easier for programmers to understand. Simplicity is composed of:

- module size
- control structure complexity
- data structure complexity
- straightforward coding

A3.2.2. Simplicity is, in some respects, the essence of maintainability. The degree of simplicity exhibited by source code is related to the understandability of the system. Maintainability is enhanced if there are fewer things the maintainer must understand or discriminate among and greater use of basic techniques and structures.

A3.3. Modularity.

A3.3.1. Modular software is composed of largely independent parts called modules. A module is defined as the smallest separately callable block of code in any language.

A3.3.2. Modularity is important because well organized software is broken down into smaller, more manageable pieces that are each more easily understood than larger, more complex code groups.

A3.4. Traceability. A module possesses the characteristic of traceability when information can be traced between associated document descriptions of the module and between documents and the module's source listing. Software may be well written and well described, but difficult to understand because of an unclear trail between top-level requirements and detailed implementation. The software maintainer must be able to trace any particular element from higher levels of program description down to executable code, and the reverse.

A3.5. Consistency.

A3.5.1. Modules are consistent when their description in the documentation agrees with preface block comments and when source listings agree with preface block comments.

A3.5.2. Consistency between documentation, preface blocks, and source listings allow the maintainer to understand the source code at the different levels of abstraction required for maintenance programming.

A3.6. Testability. A module is testable when it contains logic to recognize and appropriately handle error conditions.

A3.7. Expandability. A module is expandable when it is easy to make changes to the amounts and types of data and to the number of operations it can perform in a specified time period.

M-1

STATEMENT: The preface block contains adequate identification data..

CHARACTERISTIC: Descriptiveness

EXPLANATION: Each module should contain, within the preface block, general information about its development, including title, author, and revision history.

GLOSSARY:
PREFACE BLOCK. Separate set of contiguous comments that usually precede the main body of the software component (a similar block at any other common software component location would be satisfactory) and that provide identification information, processing information, and interface information for the software component.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Figure M-1 shows an example of header identification data that show the minimum amount of information you should expect to see.

```
/* TITLE:                          Reliability
 * MNEMONIC:                       RELIAB
 * CREATION DATE:                  15 Mar 69
 * ORIGINAL PROGRAMMER:            Af O'Tek, REL Division
 * MODIFIED BY:                    D. Gassner, 4-HD Group
 * REVISION HISTORY:
 *        12 Jun 72:    Added check for divide by zero.
 *                      Al B. Kurky, SP-48 Group
 *        14 Dec 72:    Removed unneeded check for divide by zero
 */
```

Figure M-1. Identification Data

M-2

STATEMENT: The preface block adequately describes the module's processing (including limitations of data processing).

CHARACTERISTIC: Descriptiveness

EXPLANATION: Within the preface block of each module, a statement of the module's purpose with a description of any limitations to its use (such as accuracy and timing) should be included. Special processing may be explained in the code, but should also be described in the preface block. Data processing limitations such as limits on the size of linked lists or arrays should be described.

GLOSSARY:
PREFACE BLOCK. Separate set of contiguous comments that usually precede the main body of the software component (a similar block at any other common software component location would be satisfactory) and that provide identification information, processing information, and interface information for the software component.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Figure M-2 illustrates a module-level description of processing.

```
/*******************************************************************************************
PURPOSE:        Calculate targeting delay interval.

CALCULATION:    This module calculates the targeting delay interval by determining the distance from the current
                positional vector to the input target location, and dividing by the current velocity.

ASSUMPTIONS:    Input  X > 0.0

PROCESSING:
        1. For input values of X such that 0.0 < ABS (X) <= .01, module returns X.
        2. For input values of X such that 0.01 < ABS (X) <= PI/2, module calculates the truncated series
           1/x + 1/x^2 + 1/x^3.
        3. For input values of X such that ABS (X) > PI/2, module returns with error flag set, ERFL = 1.

LIMITATIONS: Limited to five significant digits. Module is called once every 20 ms, requires
                12 ms to complete.
*******************************************************************************************/
```

Figure M-2: Module Processing

M-3

STATEMENT: The preface block adequately describes the interfaces.

CHARACTERISTIC: Descriptiveness

EXPLANATION: Each module should have a preface block that includes a description of each of the interfaces, including inputs, outputs, and calls. Data declarations may be considered a part of the preface block if they are all together near the preface block. In addition to calls, this question looks for some amount of description of each of the input and output variables. All interface information should match the interface information given in the documentation.

GLOSSARY:
INTERFACE. A shared boundary between the software and other software, hardware, or users.
PREFACE BLOCK. Separate set of contiguous comments that usually precede the main body of the software component (a similar block at any other common software component location would be satisfactory) and that provide identification information, processing information, and interface information for the software component.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Figure M-3 illustrates header information for a module interface.

```
INPUT ARGUMENTS:
    POS:    Real array; X,Y,Z positional vector (meters)
    VEL:    Real array; velocity vector (m/sec)
    ACC:    Real array; acceleration vector (m/sec^2)
    STATE:  Real array; state vector (POS, VEL, ACC)

OUTPUT ARGUMENTS:
    LAT:    Latitude (DDDMMSS)
    LONG:   Longitude (DDDMMSS)
    ITIME:  Current time (HHMMSS)
    INT:    Delay interval (HHMMSS)
    ITGT:   Time over target (HHMMSS)

CALLS TO:
    GET_LAT
    GET_LONG

CALLS FROM:
    CALC_POSITION
```

Figure M-3. Module Interface Description

M-4

STATEMENT: Identifier names are descriptive of their use.

CHARACTERISTIC: Descriptiveness

EXPLANATION: When modules and data items are named descriptively, the time it takes a maintainer to find a function and understand it are decreased. This involves more than the use of a naming convention because, by themselves, naming conventions do not necessarily ensure descriptiveness.

GLOSSARY:
DATA ITEM. Any parameter, variable, or constant.
MODULE. The smallest callable component part of a software product. A module is a logically separable part of the program.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Figure M-4 shows an Ada procedure that uses descriptive names.

```
procedure Room_Environment is

    Room :          ROOM_ID_TYPE;
    Temperature:    TEMPERATURE_TYPE ;
    Humidity:       HUMIDITY_TYPE;
    Voltage:        VOLTAGE_TYPE ;

begin
    for Room in ROOM_ID_TYPE loop
        Temperature := Thermometer_Reading(Room);
        Humidity := Hygrometer_Reading(Room);
        Voltage := Voltmeter_Reading(Room);
        Put_In_Grid(Room, Temperature, Humidity, Voltage);
    end loop;
end Room_Environment;
```

Figure M-4. Descriptive Identifiers Names.

M-5

STATEMENT: Embedded comments adequately describe local data.

CHARACTERISTIC: Descriptiveness

EXPLANATION: Local data should be described in a special "declaration" section. Depending upon the source language, the declaration statements may intrinsically describe some or all of a data item's attributes (e.g., type, units, range, description, etc.). Global data items are previously covered in the interfaces question.

GLOSSARY:
EMBEDDED COMMENTS. Comments exclusive of those in a preface block.
LOCAL DATA. Data that can be accessed by only one module or set of nested modules in a computer program.

SPECIAL RESPONSE INSTRUCTIONS: Answer N/A if there are no local data.

EXAMPLE: Figure M-5 shows an example with embedded comments that describe local data.

```
— procedure sums the ASCII codes of a sequence of input lines
— processing stops when the user enters a single carriage on a line

procedure SUM_ASCII_CODES is

        — local data
    SUM             : NATURAL := 0;   -- sum of ASCII codes
    LENGTH          : NATURAL := 1;   -- length input string
    LINE            : STRING(1..132); -- input line buffer

        -- numeric I/O instantiation
    package NUM_IO is new TEXT_IO.INTEGER_IO(INTEGER);

        -- function sums the ASCII codes of a single input line
    function LINE_SUM(LINE      : in STRING) return NATURAL is

            -- temporary variable
    RESULT      : NATURAL := 0;

begin       -- LINE_SUM
    for I in LINE'range loop
        RESULT := RESULT + character'pos(LINE(I));
    end loop;
    return RESULT;
end LINE_SUM;

begin       -- SUM_ASCII_CODES
    SUM_CODES: loop exit when LENGTH = 0;

        -- get input line
    TEXT_IO.GET_LINE(LINE, LENGTH);

        -- clear buffer
    if LENGTH = LINE'LAST then
            TEXT_IO.SKIP_LINE;
    else  -- pad input line
            LINE(LENGTH+1..LINE'LAST) := (others => ' ');
    end if;
        -- running total of ASCII codes in the input lines
    SUM := SUM + LINE_SUM(LINE => LINE(1..LENGTH));

    end loop SUM_CODES;

    NUM_IO.PUT(SUM);
    TEXT_IO.NEW_LINE;
end SUM_ASCII_CODES;
```

Figure M-5. Embedded Comments

M-6

STATEMENT: Embedded comments adequately describe the processing (exclusive of error processing) and transfers of control within this module.

CHARACTERISTIC: Descriptiveness

EXPLANATION: The understandability of tasks within a module is greatly aided by including concise, descriptive comments preceding each function. It is not necessary that each computation or control branch be commented. However, understanding the module's general processing and what conditions cause a transfer of control and to where control is transferred is important. A control statement's syntax or control parameter name may clearly explain a given transfer of control without need for further comment.

GLOSSARY:
CONTROL STATEMENT. A program statement that selects among alternative sets of program statements or affects the order in which operations are performed. For example, IF-THEN-ELSE, CASE.
EMBEDDED COMMENTS. Comments exclusive of those in a preface block.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Figure M-5 shows an example with embedded comments that described the transfer of control.

M-7

STATEMENT: Embedded comments adequately describe error processing.

CHARACTERISTIC: Descriptiveness

EXPLANATION: The module should describe any exceptions that could occur and how these exceptions are handled. The maintainer should be able to determine the output of the processing if an exception is encountered.

For error processing, look for comments describing reserved words used for error processing or special implementations of selection control structures. Any exception handling or abnormal transfer of control is adequately described.

GLOSSARY:
EMBEDDED COMMENTS. Comments exclusive of those in a preface block.
ERROR PROCESSING. The steps required to set program data and control states following the detection of an undesirable event.
EXCEPTION. An event that causes suspension of normal program execution.

SPECIAL RESPONSE INSTRUCTIONS: Answer N/A if there is no error processing.

EXAMPLE: Figure M-7 shows an example of embedded error processing comments.

```
/* if the denominator contains a zero, skip the division and avoid a divide by zero error */

if(num_bins <> o)
{
      fract_bins = total_bins / num_bins;
}
```

Figure M-7. Embedded Error Processing Comments.

M-8

STATEMENT: Embedded comments adequately describe any machine dependencies in the source listing.

CHARACTERISTIC: Descriptiveness

EXPLANATION: All uses of machine dependencies should be clearly commented within the preface block or embedded comments of the module.

GLOSSARY:
EMBEDDED COMMENTS. Comments exclusive of those in a preface block.
MACHINE DEPENDENCIES. Pertaining to software that relies on features unique to a particular type of computer and, therefore, executes only on computers of that type.
PREFACE BLOCK. Separate set of contiguous comments that usually precede the main body of the software component (a similar block at any other common software component location would be satisfactory) and that provide identification information, processing information, and interface information for the software component.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Figure M-8 shows embedded comments about machine dependencies.

```
PACKAGE Low_Level_IO IS

-- This package to be used with the Intel 8086
     Type Device_Type IS RANGE 0 .. 16#FFFF#;
     Type Byte_Type IS RANGE 0 .. 16#FF#;

END Low_Level_IO;
```

Figure M-8. Commented Machine Dependencies

M-9

STATEMENT: Statement Labels are meaningful and reasonably used.

CHARACTERISTIC: Descriptiveness

EXPLANATION: High-level languages and assemblers support different levels of labeling and whatever method used, the labels should be functional and meaningful.

GLOSSARY:
LABEL. A name or identifier assigned to a computer program statement to enable other statements to refer to that statement.

SPECIAL RESPONSE INSTRUCTIONS: Answer N/A if statement labels are not required.

EXAMPLE: For example, the FORTRAN code segment in figure M-9(a) shows a typical method using FORTRAN labels of up to five digits numbers in ascending order from the beginning to the end of the module. While this method is not necessarily self-documenting, this method is required by FORTRAN. Figure M-9(b) shows user-defined labels in Ada that are used to delineate two nested control statements. In this case, the outside label is required while the inside label is used for clarity.

```
        DO 1600 RCOUNT = 1,ROW
            DO 1500 CCOUNT = 1,COLUMN
                TEMP = TEMP*ARRAY1(RCOUNT,CCOUNT)
1500    CONTINUE
            T3 = T3*TEMP*TEMP
1600    CONTINUE
        V3 = T3/COLUMN
        DO 1800 RCOUNT = 1,ROW
            DO 1700 CCOUNT = 1,COLUMN
                TEMP = TEMP*ARRAY1(RCOUNT,CCOUNT)
1700    CONTINUE
1800    CONTINUE
```

Figure M-9(a). FORTRAN Labels

```
Line_Loop : loop  -- Process each line in the message
    Character_Loop : loop  -- Process each character in the line
        My_Int_IO.Get(Next_Code);
        exit Character_Loop when Next_Code = End_of_Line_Code;
        exit Line_Loop when Next_Code = End_Of_Message_Code;
        Write_Letter(Next_Code);
    end loop Character_Loop;
    New_Line;      -- handle the end-of-line code
end loop Line_Loop;
Put_Line("#");      -- handle the end-of-message code
```

Figure M-9(b). Ada Labeling

M-10

STATEMENT: The source code uses indentation and vertical spacing to enhance readability.

CHARACTERISTIC: Descriptiveness

EXPLANATION: There should be a consistent style for vertical spacing (white space) and indentation for control statements. Vertical spacing includes having only one executable statement per physical line of code, white space between separate control statements, and comments to separate logical blocks of code. Indentation should reflect the logical structure of control statements.

GLOSSARY:
CONTROL STATEMENT. A program statement that selects among alternative sets of program statements or affects the order in which operations are performed. For example, IF-THEN-ELSE, CASE.
STANDARDS. Mandatory requirements employed and enforced to prescribe a disciplined uniform approach to software development, that is, mandatory conventions and practices are, in fact, standards.

SPECIAL RESPONSE INSTRUCTIONS: Do not answer 5 or 6.

EXAMPLE: Figure M-10 shows a module where vertical spacing, comments, and indentation have been used. Comment lines delineate declarative regions, white space delineates logical blocks of code, and indentation shows the logical structure of the control statements.

```
-- procedure sums the ASCII codes of a sequence of input lines
-- processing stops when the user enters a single carriage on a line

procedure SUM_ASCII_CODES is

        -- local data
    SUM             : NATURAL := 0;   -- sum of ASCII codes
    LENGTH          : NATURAL := 1;   -- length input string
    LINE            : STRING(1..132); -- input line buffer

        -- numeric I/O instantiation
    package NUM_IO is new TEXT_IO.INTEGER_IO(INTEGER);

        -- function sums the ASCII codes of a single input line
    function LINE_SUM(LINE      : in STRING) return NATURAL is

            -- temporary variable
    RESULT      : NATURAL := 0;

  begin     -- LINE_SUM
    for I in LINE'range loop
        RESULT := RESULT + character'pos(LINE(I));
    end loop;
    return RESULT;
  end LINE_SUM;

begin     -- SUM_ASCII_CODES
  SUM_CODES: loop exit when LENGTH = 0;

        -- get input line
    TEXT_IO.GET_LINE(LINE, LENGTH);

        -- clear buffer
    if LENGTH = LINE'LAST then
            TEXT_IO.SKIP_LINE;
    else  -- pad input line
            LINE(LENGTH+1..LINE'LAST) := (others => ' ');
    end if;
        -- running total of ASCII codes in the input lines
    SUM := SUM + LINE_SUM(LINE => LINE(1..LENGTH));

  end loop SUM_CODES;

  NUM_IO.PUT(SUM);
  TEXT_IO.NEW_LINE;
end SUM_ASCII_CODES;
```

Figure M-10. Vertical Spacing and Indentation

M-11

STATEMENT: The module uses structured programming.

CHARACTERISTIC: Simplicity

EXPLANATION: There are two aspects of structured programming to evaluate. First, the module should be constructed of a basic set of control statements, each having one entry and one exit. The set of control statements typically includes: a sequence of two or more instructions, selection, and repetition. Second, the module should be constructed using a disciplined approach that adheres to specified rules based on top-down design or object-oriented design, stepwise refinement of data, system structures, and processing step. It should flow essentially from top to bottom. In a module, the dominant characteristic will be the use of a basic set of control statements to perform the processing. A structured programming language such as Ada or C provides mechanisms to implement structured programming. Probably the most obvious sign of unstructured programming is the use of GOTO (or the equivalent) statements.

GLOSSARY:
CONTROL STATEMENT. A program statement that selects among alternative sets of program statements or affects the order in which operations are performed. For example, IF-THEN-ELSE, CASE.
STRUCTURED DESIGN. Any disciplined approach to software design that adheres to specified rules based on principles such as modularity, top-down design, and stepwise refinement of data, system structures, and processing steps.
STRUCTURED PROGRAM. A computer program constructed of a basic set of control structures, each having one entry and one exit. The set of control structures typically includes: a sequence of two or more instructions, conditional selection of one of two or more instructions, and repetition of a sequence of instructions.
STRUCTURED PROGRAMMING. Any software development technique that includes structured design and results in the development of structured programs.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Figure M-11(a) shows a module which does not fully implement structured programming. The executable statements are sequential or conditional selection. However, no attempt has been made at structured design. There are five functionally related tasks that can be written as subprograms and a user-defined type called "FRACTION" will make the code easier to understand. These changes are implemented in figure M-11(b).

(a)

```
with TEXT_IO;
procedure UNSTRUCTURED is
      N_X, D_X, N_Y, D_Y, D_Z, N_Z : integer;
      OP : character;
      package MY_INT_IO is new
            TEXT_IO.INTEGER_IO(INTEGER);
begin
      TEXT_IO.PUT_LINE("Enter + to add fractions ");
      TEXT_IO.PUT_LINE("Enter - to subtract fractions");
      TEXT_IO.PUT_LINE("Enter * to multiply fractions ");
      TEXT_IO.PUT_LINE("Enter / to divide fractions");
      TEXT_IO.GET(OP);
      TEXT_IO.SKIP_LINE;
      TEXT_IO.PUT_LINE("Enter First Numerator:  ");
      MY_INT_IO.GET(N_X);
      TEXT_IO.SKIP_LINE;
      TEXT_IO.PUT_LINE("Enter First Denominator:  ");
      MY_INT_IO.GET(D_X);
      TEXT_IO.SKIP_LINE;
      TEXT_IO.PUT_LINE("Enter Second Numerator:  ");
      MY_INT_IO.GET(N_Y);
      TEXT_IO.SKIP_LINE;
      TEXT_IO.PUT_LINE("Enter Second Denominator:  ");
      MY_INT_IO.GET(D_Y);
      TEXT_IO.SKIP_LINE;
case OP is
      when '+' =>
            N_Z := N_X*D_Y + N_Y* D_X;
            D_Z := D_X* D_Y;
      when '-' =>
            N_Z := N_X*D_Y - N_Y* D_X;
            D_Z := D_X* D_Y;
      when '*' =>
            N_Z := N_X*N_Y;
            D_Z := D_X*D_Y;
      when '/' =>
            GOTO 10
      when others =>
            GOTO 20
      end case;

10      N_Z := N_X*D_Y;
            D_Z := D_X*N_Y;
20      TEXT_IO.PUT_LINE("invalid operation");
end UNSTRUCTURED;
```

(b)

```
with FRACTION_HANDLER;
use FRACTION_HANDLER;
with FRACTIONIO;  use FRACTIONIO;
with TEXT_IO; use TEXT_IO;

procedure STRUCTURED is
      X, Y :     FRACTION;
      OP   :     positive := 1;
begin
      Menu;
      GET_CHOICE(OP);
      X := GET_FRACTION;
      Y := GET_FRACTION;
      PERFORM_OP(OP, X, Y);
end STRUCTURED;
```

Figure M-11. Unstructured/Structured Programming

M-12

STATEMENT: The module contains a manageable number of executable statements.

CHARACTERISTIC: Simplicity

EXPLANATION: Small modules are easier to maintain than larger modules. Remember this count is for executable lines of code, not total lines of code.

GLOSSARY:
EXECUTABLE STATEMENT. A statement as defined by a high-order language excluding comments, data declarations, and variable/constant declarations.

SPECIAL RESPONSE INSTRUCTIONS: Do not answer 6.

EXAMPLE: The code segment in figure M-12 has 31 lines; however, there are only 6 executable statements (see the comments for explanation).

	Code Segment	Remarks
1	#include <stdio.h>	*lines 1-5, preprocessor commands* ***not executable***
2	#define NAME_SIZE 20	
3	#define ARRAY_SIZE 20	
4	#define row_dimension 10	
5	#define col_dimension 5	
6		*blank line,* ***not executable***
7	typedef struct NODE	*lines 7-12, type definitions* ***not executable***
8	{	
9	int num_kids;	
10	int num_rents;	
11	int value;	
12	} LIST_NODE;	*blank line,* ***not executable***
13		*type definition,* ***not executable***
14	typedef LIST_NODE GRID[row_dimension][col_dimension];	*blank line,* ***not executable***
15		*function declaration,* ***not executable***
16	void displayGrids(GRID old_node)	*delimiter,* ***not executable***
17	{	*variable declaration,* ***not executable***
18	int row = 0;	
19		***executable***
20	printf("\n\tOLD\t\t\t\t\tNEW\n");	*comment,* ***not executable***
21	// Iterate through entire grid print the values	***executable***
22	for(row=1; row<=3; ++row)	*delimiter,* ***not executable***
23	{	***executable***
24	printf("%d %d %d,"	*lines 25-27, extension of line 24, count as a*
25	old_node[row][1].value,	*single* ***executable*** *statement*
26	old_node[row][2].value,	
27	old_node[row][3].value);	*delimiter,* ***not executable***
28	}	***executable***
29	printf("\nHit return to continue... \n");	***executable***
30	getchar();	*delimiter,* ***not executable***
31	}	

Figure M-12. Executable Lines of Code

M-13

STATEMENT: The number of operators in this module is manageable.

CHARACTERISTIC: Simplicity

EXPLANATIONS: The concept is that the more operators there are, the more discriminations one must make to understand the module's function. Again, a count is to be made. It is not intended that the evaluator spend a great amount of time counting operators. In a few minutes using the guidelines below, it should be possible to get a reasonable estimate of the number of operators. For all questions addressing size simplicity, the underlying concept is that a small module is characteristically easier to maintain than a large module. The following are guidelines of what is an operator. Some examples are also provided. The following list should be considered representative and not complete:

1. All typical language verbs are operators:
   - e.g., unary op: negation (-), set complement (')
   - binary op: addition (+) subtraction (-)
     multiplication (*) division (/)
     exponentiation (**) assignment (=)
   - relation op: less than (LT, <)
     greater than (GT, >)
     equal (EQ, =)
     not equal (NE, #)
     less than or equal (LE, <=)
     greater than or equal (GE, >=)
   - logical op: AND, OR, NOT
   - control op: decision (IF THEN ELSE, CASE)
     iterative (DO loop, DOWHILE, DOUNTIL, etc.)
     sequential grouping (BEGIN, END, DO)
     GOTO (each distinct GOTO label is a unique operator)

GLOSSARY:
OPERATOR. A mathematical or logical symbol that represents an action to be performed in an operation. For example, in the expression A = B + 3, + is the operator representing addition and the = is the operator representing assignment.

SPECIAL RESPONSE INSTRUCTIONS: Do not answer 6.

EXAMPLE: None

M-14

STATEMENT: The module contains a manageable amount of branching.

CHARACTERISTIC: Simplicity

EXPLANATION: The amount of branching within a module is strongly related to its cyclomatic complexity. Modules with fewer branches or paths are simpler to maintain. If you have complexity metrics available, use this information to help evaluate this statement.

GLOSSARY:
BRANCHING. Nonsequential execution based on control statements (e.g., IF-THEN, looping, or CASE) or explicit transfers of control (e.g., GOTO).

SPECIAL RESPONSE INSTRUCTIONS: Do not answer 6.

EXAMPLE: The simple code segment in figure M-14(a) shows a selection statement with six branches that is written improperly. First, using only IF-THEN statements each of the conditional statements will be evaluated, using IF-THEN-ELSE statements the selection would short-circuit at the first condition that becomes true. Second, the outer most IF-THEN should handle the condition when the character is an upper case. Third, the selection statements should have a default condition for when the character is lower case, but greater than "e." These problems, taken with the large amount of branching makes a control statement like this difficult to maintain.

Figure M-14(b) illustrates how this code segment could be rewritten which eliminates the listed problems. The selection of the correct case is more efficient, the case of an invalid input is taken care of, and the default condition for valid input is handled. So, even though there are still six branches, the second code module is much easier to maintain.

NOTE: the simplest code to maintain is straight sequential code or no branching at all.

(a)

```
procedure Selection(CH : in character) is
begin       --process lower case only
  if (CH >= 'a' and CH <= 'z') then
       if( CH = 'a') then
            PUT_LINE("argali, a sheep of Asia");
       end if;
       if(CH = 'b') then
            PUT_LINE("babirusa, a pig of Malya");
        end if;
        if(CH = 'c') then
            PUT_LINE("coati, racoon like mammal");
        end if;
        if(CH = 'd') then
            PUT_LINE("desma, acquati ccritter");
        end if;
        if(CH = 'e') then
            PUT_LINE("echidna, the anteater");
        end if;
  end if;
end Selection;
```

(b)

```
procedure Selection(CH : in character) is
begin
  if (CH >= 'a' and CH <= 'z')
    then --process lower case only
      case CH is
        when 'a' => PUT_LINE("argali, a sheep of
                                        Asia");
        when 'b' => PUT_LINE("babirusa, a pig of
                                        Malya");
        when 'c' => PUT_LINE("coati, racoon like
                                        mammal");
        when 'd' => PUT_LINE("desma, acquatic
                                        critter");
        when 'e' => PUT_LINE("echidna, the
                                        anteater");
        when others => PUT_LINE("I'm stumped");
      end case;
    else
      PUT_LINE("I only process lower case
                              letters");
    end if;
end Selection;
```

Figure M-14. Selection Code

M-15

STATEMENT: The module contains manageable levels of nesting.

CHARACTERISTIC: Simplicity

EXPLANATION: Modules with fewer levels of nesting are easier to understand and maintain. In some cases, the algorithms required to solve problems may require many levels of nesting. Although the only solutions to their particular problems, these algorithms will be more difficult to maintain than other modules with less nesting.

GLOSSARY:
NESTING. Incorporating a control statement or control statements into another control statement.

SPECIAL RESPONSE INSTRUCTIONS: Do not answer 6.

EXAMPLE: Figure M-15 shows a module that contains three levels of nesting. Nested For-Loops are the typical method for accessing each element in a two-dimensional array and this level of nesting should be expected and is manageable. However, levels 2 and 3 are nested conditional statements within the nested For-Loops and without the embedded comments this code segment may be difficult to understand.

NESTING LEVEL	MODULE
	/************************ flip_value_status ****************************** This routine is called in order to flip the value of a structure field in a grid by accessing each element of the grid in row major form. If the tolerance is within limits, the grid cell structure field "value" is flipped ***/ void flip_value_status(NODE_ARRAY grid, int err, int tolerance) {
0	int num=0,row=0,col=0;
	for(row=1; row<=15; ++row) /* access each row */
1	{
	for(col=1; col<=15; ++col) /* access each column in the row */
2	{
	if(tolerance < err) /* if within tolerance, flip the value */
3	{

	if(grid[row][col].value==ACTIVE)
	grid[row][col].value=INACTIVE;
	else
	grid[row][col].value=ACTIVE;

	}
	}
	}
	} /* END flip_value_status */

Figure M-15. Nested Control Structures.

M-16

STATEMENT: The module contains a manageable number of unique data items.

CHARACTERISTIC: Simplicity

EXPLANATION: Consider both the number of unique data items and their complexity. It should be relatively easy to obtain a reasonable estimate of the number of unique data items. Single instances of compound data structures are counted as a single data item. Evaluate the complexity of the data items since a small number of data items may be unmanageable. This statement includes both local and global data items.

GLOSSARY:
DATA ITEM. Any parameter, variable, or constant.

SPECIAL RESPONSE INSTRUCTIONS: Do not answer 6.

EXAMPLE: The example in figure M-16 shows what appears to be a small, simple module with only two data items (xdrs and objp), so the number of data items seems to be manageable. However, the pointer parameter ***objp*** is being cast to a pointer to a pointer to a character. Since there are no embedded comments that explain the processing, the use of this single data item makes this module unmanageable.

```
boot_t xdr_MsgProfileToggle(XDR *xdrs,  MsgProfileTogglePtr *objp)
{
if(!xdrs_pointer(xdrs, (char **)objp, sizeof(MsgProfileToggle), xdr_MsgProfileToggle)) {
      return (FALSE);
}
return (TRUE);
}
```

Figure M-16. A Single, Unmanageable Data Item.

M-17

STATEMENT: This module uses a manageable number of composite data structures.

CHARACTERISTIC: Simplicity

EXPLANATION: Data structures are either scalar data types or composite data types. Composite data structures are composed of other scalar or composite data types. In general, composite data structures are more complex than scalar data types, but if used properly they can make the source code easier to understand and modify. Rather than the number of composite data structures, the evaluator should consider whether the use of composite data structures simplifies the module's processing and makes software maintenance easier. In a good design, the data structures used should reflect the functional use of the data and hence simplify the programming task.
This question is directed at all data structures used within the module, including global data structures.

GLOSSARY:
COMPOSITE DATA TYPE. A data type, each of whose members are composed of multiple data items. For example, a data type called PAIRS whose members are ordered pairs (x,y).
SCALAR DATA TYPE. A data type that represents a single element of information.

SPECIAL RESPONSE INSTRUCTIONS: Do not answer 6.

EXAMPLE: Arrays, records (structures), unions, linked lists, search trees are examples of composite data structures. The first code segment in figure M-17(a) shows a composite data structure that might be difficult to maintain. The PTR_ARRAY contains a list of void pointers and from this example a maintainer can only guess that the pointers will be cast to character pointers and strings will be stored in the arrays. The second code segment in figure M-17(b) shows how this structure can be written without ambiguity. ***NOTE***: The programmer may have a valid reason to write the structure shown in figure M-17(a) but the documentation should describe the reasons for this approach.

(a)

```
#define name_size 21
#define array_size 10
typedef void *VPTR;
typedef VPTR PTR_ARRAY[array_size];
typedef struct node
{
    char        name[name_size];
    int         value;
    PTR_ARRAY   rents;
    PTR_ARRAY   kids;
    int         num_kids;

} NODE;
typedef NODE *NODE_PTR;
```

(b)

```
#define NAME_SIZE    21
#define STRING_SIZE  10

typedef struct NODE
{
    char    name[NAME_SIZE];
    char    rents[STRING_SIZE];
    char    kids[STRING_SIZE];
    int         num_kids;
    int         num_rents;
    int         value;

} LIST_NODE, *NODE_PTR;
```

Figure M-17. Composite Data Structures.

M-18

STATEMENT: The module's use of complex expressions is manageable.

CHARACTERISTIC: Simplicity

EXPLANATION: While sometimes necessary, complex expressions are difficult to understand. However, spacing, indentation, and parentheses can help the maintainers understand the code and expressions. Consider complex expressions used in formulae as well as to control branching.

GLOSSARY:
COMPLEX EXPRESSION. A statement or declaration that expresses a formula for calculating a value. A complex expression is considered difficult, and composed of expressions and operators.

SPECIAL RESPONSE INSTRUCTIONS: Do not answer 6.

EXAMPLE: Figure M-18(a) shows a complex expression that may be used to expand referencing to a two-dimensional array. This form is more efficient since it uses only integer arithmetic and fewer machine language instructions are required; however, it is difficult to interpret. Figure M-18(b) is a code segment that performs the same function and is easy to understand and should be used if there are no overriding space or time constraints.

(a) Complex

```
void test(void)
{
    int   test_array[2][3] ={ {0, 1, 2},
                              {3, 4, 5}
                            };

    *((int *)((char *) test_array + (1*3*4)) + (2*4)) = 5;

}
```

(b) Not Complex

```
void test(void)
{
    int   test_array[2][3] ={ {0, 1, 2},
                              {3, 4, 5}
                            };

    test_array[1][2] = 5;

}
```

Figure M-18. Complex Expressions.

M-19

STATEMENT: There is no esoteric programming in this module.

CHARACTERISTIC: Simplicity

EXPLANATION: Esoteric programming techniques increase the workload on the maintainers by requiring more time to try to figure out what the originator of the code was trying to do.

GLOSSARY:
ESOTERIC. Designed for or understood by few.

SPECIAL RESPONSE INSTRUCTIONS: Do not answer 5 or 6.

EXAMPLE: Figure M-19(a) shows a matrix being initialized. However, when ROW > COL then COL/ROW = 0, due to intrinsic FORTRAN integer division rules and when COL > ROW then ROW/COL = 0. The net result is that all diagonal elements, ARRAY1(ROW,COL), of the matrix are assigned the value 1.0 and all other elements are assigned the value 0.0. Figure M-19(b) shows a more understandable and more efficient version.

(a)

```
      DO 10 ROW=1, SIZE
         DO 10 COL=1, SIZE
10             ARRAY1(ROW, COL) =
                  (ROW/COL)*(COL/ROW)
```

(b)

```
      DO 20 ROW=1, SIZE
         DO 10 COL=1, SIZE
10             ARRAY1(ROW,COL) = 0.0
20       ARRAY1(ROW,ROW) = 1.0
```

Figure M-19. Esoteric Programming

M-20

STATEMENT: This module contains no extraneous code.

CHARACTERISTIC: Simplicity

EXPLANATION: Extraneous or dead code makes modules more difficult to trace and understand. If code has been commented-out and its purpose is not clearly defined, treat those statements as extraneous code.

GLOSSARY:
EXTRANEOUS CODE. Code not used or that can never be executed, variables never referenced, types not used, or included software components never used.

SPECIAL RESPONSE INSTRUCTIONS: Do not answer 5 or 6.

EXAMPLE: The Pascal code segment in figure M-20(a) shows a variable being tested shortly after it was set and the writeln statement after the test will never be executed. Figure M-20(b) shows a FORTRAN branch statement that jumps to the very next line, which is executed next anyway.

(a)

```
    IntegerInput := 4;
    RealInput := 3.14159;
    CharacterInput := 'Z';

    if (IntegerInput > 7) then
      writeln('Integer input out of range.');

(* The above 'writeln' statement is extraneous  *)
(* and will never be executed, since            *)
(* IntegerInput = 4 *)
```

(b)

```
      IF (A.LT.B) GOTO 10
      IF (A.LT.C) GOTO 10
            MAX = A
            GOTO 30
10    IF (B.LT.C) GOTO 20
            MAX = B
            GOTO 30
20    MAX = C
      GOTO 30
    C This 'GOTO 30' statement is extraneous.
30    PRINT *, MAX
```

Figure M-20. Extraneous Code.

M-21

STATEMENT: This module uses a manageable amount of machine-dependent techniques or language extensions.

CHARACTERISTIC: Simplicity

EXPLANATION: Machine-dependent coding techniques refer to functions specific to a particular computer architecture. Examples of these are hardware interrupts and bit/byte manipulations. Language extensions are often provided by compiler vendors to add capabilities to older languages such as FORTRAN. Machine-dependent coding techniques and language extensions require the maintainer to have specialized knowledge. Be aware that cases exist where machine-dependent code is required and may make the module's processing simpler.

GLOSSARY:
LANGUAGE EXTENSION. Feature in a programming language that is not in the language's standard (i.e., ANSI Standard, ISO Standard.) For example, Microsoft C++ has the statements CIN and COUT that are not part of C++'s ANSI Standard.
MACHINE DEPENDENCIES. Pertaining to software that relies on features unique to a particular type of computer and, therefore, executes only on computers of that type.

SPECIAL RESPONSE INSTRUCTIONS: Do not answer 5 or 6.

EXAMPLE: Figure M-21 shows Pascal code used to enable asynchronous communication on an IBM PC. The constants (e.g., IER or MCR) are specific to the IBM architecture. Bit masking is also performed.

```
PORT[PICMSK] := PORT[PICMSK] and imvalue;   { Enable ASynch Int}
 PORT[IER+ComBase] := $01;                  { Enable some interrupts}
 PORT[MCR+ComBase] := $0B;                  { Set RTS, DTR, OUT2}
 LCRreg := $80;                             { Set Divisor Latch Access Bit in LCR }
 LCRreg := LCRreg or paritycode[Parity];    { Setup Parity}
 LCRreg := LCRreg or databitscode[Databits];{ Setup # data bits}
 LCRreg := LCRreg or stopbitscode[Stopbits];{ Setup # stop bits}
 PORT[LCR+ComBase] := LCRreg;               { Set Parity, Data and Stop Bits
                                              and set DLAB}
 PORT[DLM+ComBase] := Hi(baudcode[Baud]);   { Set Baud rate}
 PORT[DLL+ComBase] := Lo(baudcode[Baud]);   { Set Baud rate}
 PORT[LCR+ComBase] := LCRreg and $7F;       { Reset DLAB}
```

Figure M-21. Machine-Dependent Code.

M-22

STATEMENT: This module exhibits loose coupling to other modules.

CHARACTERISTIC: Modularity

EXPLANATION: Coupling is a description of the strength of association between different modules. Coupling depends on the interface complexity between modules, the point which entry or reference is made to a module, and what data passes across the interface. High coupling is a negative characteristic of a system since an attempt to modify one module may result in maintainer affecting other modules without being aware of the interaction between the modules.

GLOSSARY:
COUPLING. The manner and degree of interdependence between software components. Types include common-environmental coupling, data coupling, hybrid coupling, and pathological coupling.
INPUTS. (1) Pertaining to data received from an external source. (2) Pertaining to a device, process, or channel involved in receiving data from an external source. (3) To receive data from an external source. (includes values returned from function calls, parameters, or any global data) (4) To provide data from an external source.
OUTPUTS. (1) Pertaining to data transmitted to an external destination. (2) Pertaining to a device, process, or channel involved in transmitting data to an external destination. (3) To transmit data to an external destination.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Figure M-22(a) shows different types of coupling that can occur. Figure M-22(b) shows the varying degrees of coupling. This scale is nonlinear and data coupling (low coupling) is much better than control coupling, which is far better than content coupling (high coupling).

TYPE	LEVEL OF COUPLING
Data Coupling	One module serves as input to another module.
Stamp Coupling	When a portion of data structure, rather than simple arguments, is passed via a module interface.
Control Coupling	One module communicates information to another module for the explicit purpose of influencing the second module's execution.
External Coupling	Modules are tied to an environment external to software, such as I/O couples to peripheral devices.
Common Environment Coupling	Two modules access a common data area.
Pathological Coupling	One software module affects or depends upon the internal implementation of another.
Content Coupling	All the contents of one module are included in the contents of another module.

Figure M-22(a). Types of Coupling.

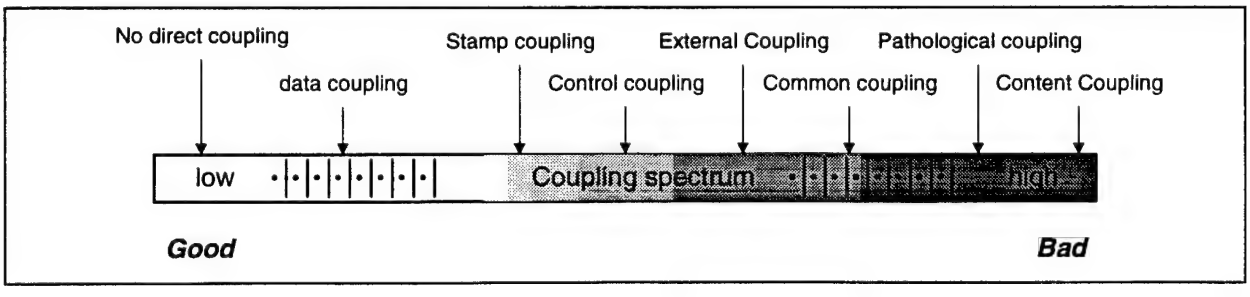

Figure M-22(b). Coupling Spectrum.

M-23

STATEMENT: This module exhibits a high level of cohesion.

CHARACTERISTIC: Modularity

EXPLANATION: Cohesion describes the strength of association of the elements ***inside*** a module. All elements of module should be strongly associated (highly cohesive) and involved in performing the same task. A cohesive module performs a single task within a software module, requiring little interaction with modules in the program. Ideally, a cohesive module should do just one thing.

GLOSSARY:
COHESION. The manner and degree to which the tasks performed by a single software component are related to one another. Types include coincidental, communicational, functional, logical, procedural, sequential, and temporal.

SPECIAL RESPONSE INSTRUCTIONS: Do not answer 6.

EXAMPLE: Figure M-23(a) shows different types of cohesion. Figure M-23(b) shows a cohesion spectrum from "bad" to "good." The scale is nonlinear and low-end cohesiveness is much worse than middle range, which is nearly as "good" as high-end cohesion.

TYPE	LEVEL OF COHESION
Coincidental	Tasks performed by a module have no functional relationship to one another.
Logical	Tasks performed by a module perform logically similar functions; for example, processing of different types of input data
Temporal	The tasks performed by a software module are all required at a particular phase of program execution; for example, a module containing all of a program's initialization tasks.
Procedural	Tasks performed by a module all contribute to a given program, such as an iteration or decision process.
Communicational	Tasks performed by a module use the same input data or contribute to producing the same output data.
Sequential	The output of one task performed by a module serves as input to another task performed within the module
Functional	Tasks performed by a module all contribute to the performance of a single function.

Figure M-23(a). Types of Cohesion

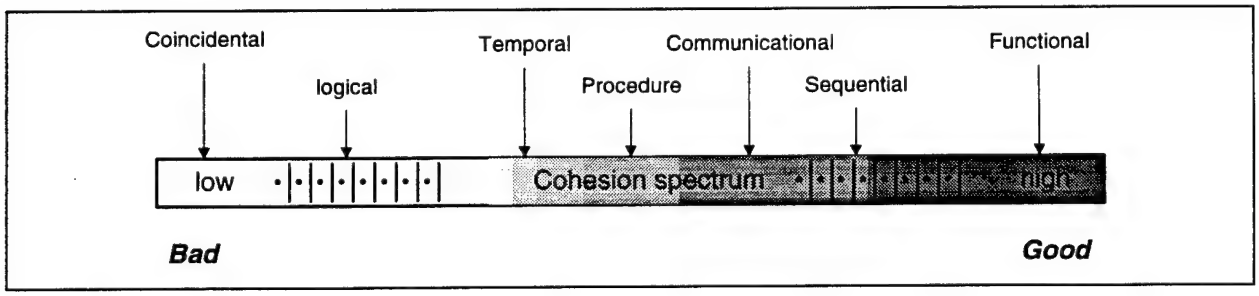

Figure M-23(b). Cohesion Spectrum

M-24

STATEMENT: Functionally related data elements within this module are organized into logical data structures.

CHARACTERISTIC: Modularity

EXPLANATION: Base your response upon how easy it is to determine the purpose of the data and how is has been organized. There may be a physical grouping of data even though it is unclear how the data are related.

GLOSSARY:
DATA STRUCTURE. A physical or logical relationship among data elements, designed to support specific data manipulation functions.
LOCAL DATA. Data that can be accessed by only one module or set of nested modules in a computer program.

SPECIAL RESPONSE INSTRUCTIONS: Answer N/A if there are no local or global data which should be logically grouped.

EXAMPLE: Suppose a test aircraft sortie program needs to record the date, aircraft, and mission flown, as well as the track the aircraft flew. Up to 10,000 data points are recorded for each sortie. The aircraft instrumentation pod records the time, latitude, longitude, altitude, airspeed, pitch, and roll. In Ada, the data structure might look like the examples in figure M-24.

(a)

```
type Track_Record is record
   Time       : ZULU_Time;
   Latitude   : Coordinates;
   Longitude  : Coordinates;
   Altitude   : Feet_AGL;
   Airspeed   : KNOTS;
   Pitch      : Degrees;
   Roll       : Degrees;
end record ;
```

(b)

```
type Sortie_Record is record
   Day        : Day_Type;
   Month      : Month_Type;
   Year       : Year_Type;
   Aircraft   : Aircraft_Type;
   Tail       : Tail_Numbers;
   Mission    : Mission_Type;
   Target     : Target_List;
   Track      : Track_Record;
end record ;
```

Figure M-24. Logical Data Structures

M-25

STATEMENT: The use of global data by this module is not excessive.

CHARACTERISTIC: Modularity

EXPLANATION: Use of global data decreases the independence of modules. Change activity and errors are more likely to propagate across modules wherever global constants or variables occur. Global data usage refers specifically to global constants or variables referenced in source code statements. This question addresses only global data that are referenced by this module.

GLOSSARY:
GLOBAL DATA ITEM. Data that can be accessed by two or more nonnested modules of a computer program without being explicitly passed as parameters between the modules.

SPECIAL RESPONSE INSTRUCTIONS: Answer 6 if there are no global data used.

EXAMPLE: Global data items are identifiable as members of such groupings as COMMON blocks, COMPOOLs, and package specifications.

M-26

STATEMENT: It is easy to trace from this module's source listing to its useful detailed description in the documentation.

CHARACTERISTIC: Traceability

EXPLANATION: All information about this module's source listing must also be easily found in the documentation to facilitate changes. Candidate information should include descriptions of complex algorithms.

GLOSSARY:
ALGORITHM. (1) A finite set of well-defined rules for the solution of a problem in a finite number of steps; for example, a complete specification of the sequence of arithmetic operations for evaluating sin x to a given precision. (2) Any sequence of operations for performing a specific task.

SPECIAL RESPONSE INSTRUCTIONS: Don't answer 6.

EXAMPLE: When looking for source code descriptions, the maintainer should be able to search the documentation, traceability matrix, on-line index, or other resource and find the desired information.

M-27

STATEMENT: Data items used in this module can be traced to their useful description in the documentation.

CHARACTERISTIC: Traceability

EXPLANATION: It is useful for the maintainer to have a link from the data used in this module to the detailed description of data in the documentation. Global data and parameters should be described in a data dictionary resource. Local data may also appear in the data dictionary.

GLOSSARY:
DATA ITEM. Any parameter, variable, or constant.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: SET/USED listings, relationship matrix, data dictionary, etc.

M-28

STATEMENT: Database items used in this module can be traced to their useful description in the database documentation.

CHARACTERISTIC: Traceability

EXPLANATION: In systems that utilize databases, software maintainers need the capability to link database elements used in a module to their detailed description in the documentation. In this sense, database elements are similar to global data items. This question is separated from the traceability of other data items because a database is different from executable source code and constitutes a very large portion of the some systems' software. Consider any formal database and any logical grouping of permanently stored data that functions as a database.

GLOSSARY:
DATABASE. A collection of interrelated data stored together in one or more computerized files.

SPECIAL RESPONSE INSTRUCTIONS: Answer N/A if no database exists.

EXAMPLE: A maintainer should be able to trace the *altitude* element of a satellite database used in a module to its description, definition, units, range information, table names, and key information in the database documentation.

M-29

STATEMENT: The information in the preface block is useful and consistent with the associated source code.

CHARACTERISTIC: Consistency

EXPLANATION: Preface block data are extremely helpful to the maintainer in understanding the code, but to be helpful the data must be accurate and contain all required information.

GLOSSARY:
PREFACE BLOCK. Separate set of contiguous comments that usually precede the main body of the software component (a similar block at any other common software component location would be satisfactory) and that provide identification information, processing information, and interface information for the software component.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: The following examples are not exhaustive:

- Inputs described agree with inputs as coded.
- Outputs described agree with outputs as coded.
- Processing information matches the processing within the unit.
- Calls described agree with calls as coded.
- Error handling described agrees with the error handling within the code.

M-30

STATEMENT: The description of the inputs and outputs in the documentation is useful and is consistent with the module's source listing.

CHARACTERISTIC: Consistency

EXPLANATION: It is important for each module and its documentation to be in agreement with each other. This question is specifically directed at agreement between the source code and documentation.

GLOSSARY:
INPUTS. (1) Pertaining to data received from an external source. (2) Pertaining to a device, process, or channel involved in receiving data from an external source. (3) To receive data from an external source. (includes values returned from function calls, parameters, or any global data) (4) To provide data from an external source.
OUTPUTS. (1) Pertaining to data transmitted to an external destination. (2) Pertaining to a device, process, or channel involved in transmitting data to an external destination. (3) To transmit data to an external destination.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: None

M-31

STATEMENT: The description of this module's control flow and processing in the documentation is useful and consistent with this module's source listing.

CHARACTERISTIC: Consistency

EXPLANATION: It is important for each module and its documentation to be in agreement with each other. Personnel who desire to know how data are transformed within the module rely upon the documentation to be accurate and complete. This question is specifically directed at agreement between source code and documentation.

GLOSSARY:
CONTROL FLOW. The sequence in which operations are performed during the execution of a computer program.
CONTROL FLOW DIAGRAM. A diagram that depicts the set of all possible sequences in which operations may be performed during the execution of a system or program. Types include box diagram, flowchart, input-process-output chart, state diagram.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Figure M-31(a) shows a code segment with the associated flow diagram in figure M-31(b). The flow diagram in the design documentation should match the control flow within the code segment.

(a)

```
/*************************************
*******

Purpose:  Accepts a character string, stores
its length into hlen; then changes every zero
(0) character to a
period (.) and every one (1) character to a
pound sign (#).

*************************************
******/
#include <string.h>

void period_to_pound(char *out_string)
{
      int  i= 0;
      int  hlen   = strlen(out_string);

      for(i=0; i<=hlen; ++i)
      {
            if(out_string[i]= '0')
                  out_string[i] = '.';

            if(out_string[i]== '1')
                  out_string[i] = '#';
      }
}
```

(b)

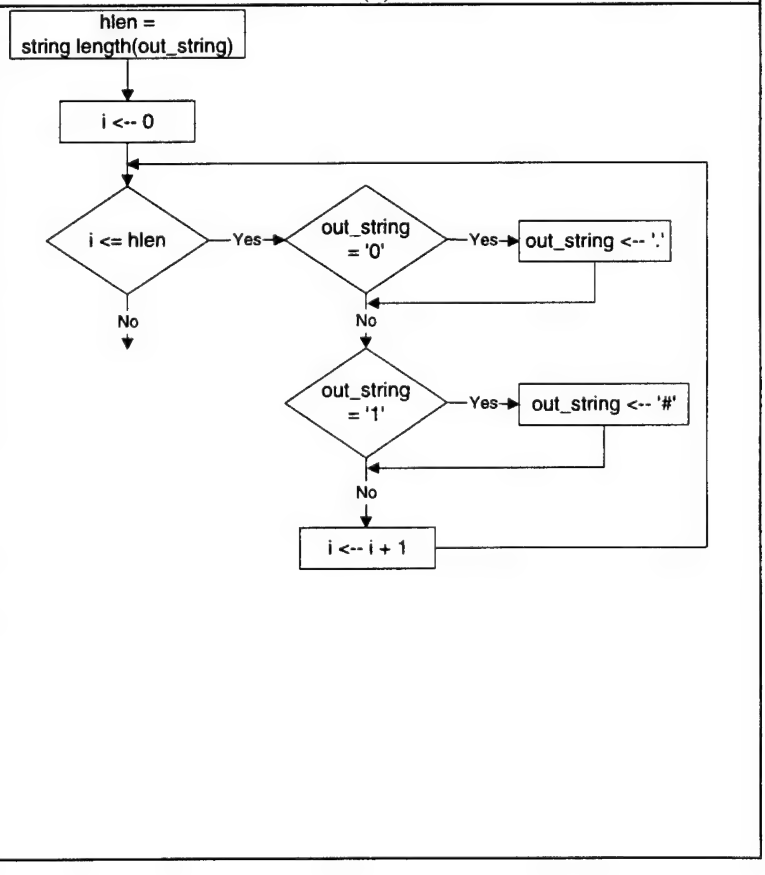

Figure M-31. Module Logic Flow.

M-32

STATEMENT: When an invalid input to the module is encountered, an appropriate action is taken.

CHARACTERISTIC: Testability

EXPLANATION: All input data should be checked as appropriate for format, range, or any other attribute that might make the data invalid. It is especially important to validate input data that are logic control parameters (e.g., decision parameter, loop index). The validation process should include the appropriate diagnostic messages and/or error codes.

GLOSSARY:
INPUTS. (1) Pertaining to data received from an external source. (2) Pertaining to a device, process, or channel involved in receiving data from an external source. (3) To receive data from an external source. (includes values returned from function calls, parameters, or any global data) (4) To provide data from an external source.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: The program in Figure M-32 reads integers from the standard input file and writes their sum to the standard output file. Invalid characters are reported on the standard error file and passed over. The standard input file is assumed to consist of a single page.

```
with TEXT_IO; use TEXT_IO;
procedure Print_Sum is
    package Count_IO is new TEXT_IO.INTEGER_IO(INTEGER);
    use Count_IO;
    Sum             : Integer := 0;
    Next_Integer    : Integer;
    C                       : Character;
begin
    while not End_Of_File Loop
        begin
            Get(Next_Integer);
            Sum := Sum + Next_Integer;
        exception
            when Data_Error =>
                Set_Output(Standard_Error);
                Put("****** Illegal character on line");
                Put(Line, Width => 0);
                    -- first parameter is a function call on Text_IO.line
                Get(C);    -- read the offending character
                Put_Line(": ' " and C and " ' ");
                Set_Output(Standard_Output);
            when End_Error =>
                Put_Line (Standard_Error, "Trailing Spaces or terminators ignored.");
                -- Now End_Of_File is true and loop will terminate
        end; --exception
    end loop;

    Put("The Sum is ");
    Put(Sum, Width => 0);
    Put_Line("");
end Print_Sum;
```

Figure M-32. Illegal Input Handling.

M-33

STATEMENT: When an internal error is encountered by the module, an appropriate action is taken.

CHARACTERISTIC: Testability

EXPLANATION: Internal errors are introduced as a result of the module's processing rather than by inputs.

GLOSSARY:
ARRAY. An n-dimensional ordered set of data items identified by a single name and one or more indices, so that each element of the set is individually addressable. For example, a matrix, table or vector.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: One common error is accessing an array subscript out of bounds and most high-level languages will check for possible out-of-bounds conditions at compile time. However, C compilers are not required to check array bounds and it is up to the programmer to perform bounds checking. For example, in figure 33-1(a) the array external A[ ] is defined in another module and the size is not specified in this module. However, the function sum(int n) returns the first N elements of the array. In this module, there is no guarantee that N is inside the bounds of the array. Figure 33-1(b) shows a common way to pass arrays to functions, along with the length, so that the bounds can be checked. Figure 33-2(a) shows another common error where an external file is opened without checking to see if the file opened properly. Figure 33-2(b) shows a common check to ensure that the file is opened properly.

(a)

```
extern A[];

int sum(int N)
{
        int i, s = 0;
        for (i = 0; i < N; i++)
                s += a[i];
        return s;
}
```

(b)

```
int SumArray(int Array[], int Array_Length)
{
        int i, s = 0;
        for (i = 0; i < Array_Length; i++)
                s += Array[i];
        return s;
}
```

Figure 33-1. Array Bound Checking.

(a)

```
#include <stdio.h>

void main(int argc, char *argv[])
{
        FILE *FP = fopen(argv[1], "r+");
        //file processing begins here
}
```

(b)

```
char stream_stat(FILE *FP);
void main(int argc, char *argv[])
{
        FILE *FP = fopen(argv[1], "r+");
                //check if error opening file
        if (stream_stat(FP) != 0)
    {       //exit if file error
                printf("Error opening %s," argv[1]);
                exit(0);
    }
        //file processing begins here
}
char stream_stat(FILE *FP)
{
        char stat = 0;
        if(ferror(FP))
                stat |= ERR_FLAG;
        if(feof(FP))
                stat |= EOF_FLAG;
        clearerr(FP);
        return stat;
}
```

Figure 33-2. Checking File Open.

M-34

STATEMENT: Constants and data structure dimensions in this module are appropriately parameterized.

CHARACTERISTIC: Expandability

EXPLANATION: By parameterizing constants and data structure dimensions, any occurrence may be modified by simply changing the parameter. In contrast, if a value is given for the constant or dimension, every occurrence of the value must be found and modified whenever it is changed. If appropriate parameterization doesn't exist, answer based on the impact to the expandability of the module.

GLOSSARY:
DATA STRUCTURE. A physical or logical relationship among data elements, designed to support specific data manipulation functions.
PARAMETERIZED. Referenced by name, not by an actual value (e.g., PI instead of 3.1415926).

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Figure M-34 shows an Ada example where the data structure dimensions are parameterized. If the dimensions need to be changed, only the constant declaration needs to be changed and if the table was not parameterized, every reference to the dimension 20 would have to be found and changed.

```
MAX_TABLE_SIZE:      constant Integer := 20;
type TABLE_ROW is array(1..MAX_TABLE_SIZE);
type TABLE_COL is array(1..MAX_TABLE_SIZE);
type TABLE_LOCATION is
      record
x_coordinate: TABLE_ROW;
y_coordinate: TABLE_COL;
end record;
.
.
--check to see if requested location is within limits
if (Location_Requested > MAX_TABLE_SIZE) then
valid_location := false;
else
      valid_location = true;
.
.
.
--zero out the Location table after using it
for I in 1..MAX_TABLE_SIZE loop
x_coordinate(I) := 0;
y_coordiante(I) := 0;
end loop;
```

Figure M-34. Parameterized Dimensions.

M-35

STATEMENT: Overall, it appears that this module's maintainability characteristics contribute to the maintainability of the system.

CHARACTERISTIC: General

EXPLANATION: Consider all aspects of maintainability including any other aspects that were not covered in this questionnaire. In addition to a 6 through 1 answer, provide any comments that reflect the overall maintainability of the module.

GLOSSARY: None

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: None

COMPUTER SOFTWARE UNIT (CSU)-LEVEL QUESTIONS

The questions listed within this section are used to evaluate the overall format and content of the source code for the computer software units being evaluated and to evaluate the consistency between the documentation and source listings. The ordering of the questions is from levels of lesser detail to higher detail. For example, the evaluator is asked to review the preface block of the CSU and its interfaces before looking at the internal parts of the CSU. An effort was also made to consolidate aspects of the CSU evaluation, such as data, at one time.

The CSU characteristics evaluated are Modularity, Descriptiveness, Consistency, Simplicity, Expandability, and Traceability (see figure A4-1). Each question in the questionnaire is used to evaluate an attribute of one of these characteristics.

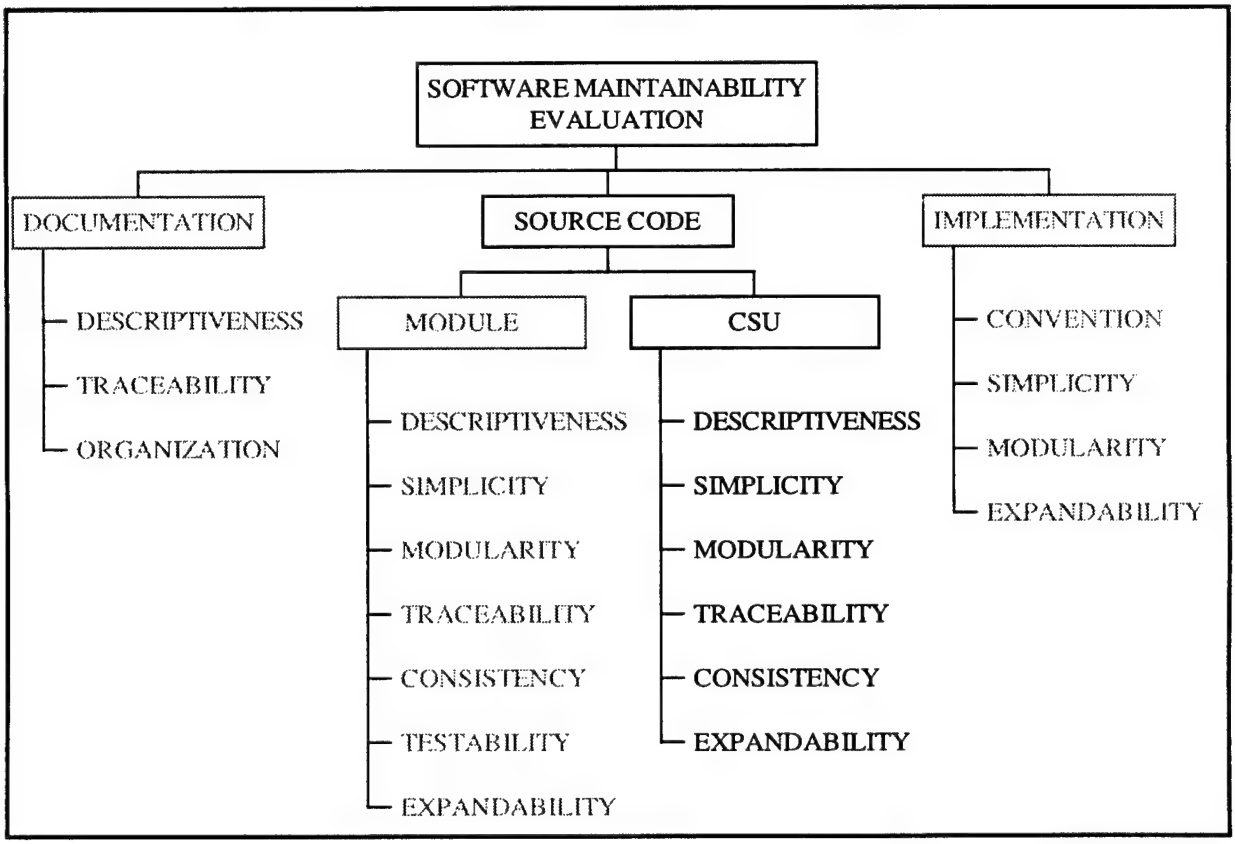

Figure A4.1. Computer Software Unit (Source Code) Characteristics.

A4.1. Descriptiveness.

A4.1.1. CSUs are descriptive when they contain information about its objectives, assumptions, constraints, inputs, processing, outputs, components, relationships to other code units, and revision status.

A4.1.2. This characteristic is very important in understanding software. The aims, assumptions, inputs/outputs, etc., are useful (in varying degrees of detail) in source listings. In addition, the descriptiveness of the source language syntax and the judicious use of embedded commentary greatly aid efforts to understand the program operation.

A4.2. Simplicity.

A4.2.1. Software source code possesses the characteristic of simplicity to the extent it lacks complexity. The aspects of complexity emphasized in the evaluation relate primarily to the concepts of size and primitives.

A4.2.2. Simplicity is, in some respects, the essence of maintainability. The degree of simplicity exhibited by source code is related to the understandability of the system. Maintainability is enhanced if there are fewer things the maintainer must understand or discriminate among and greater use of basic techniques and structures.

A4.3. Modularity.

A4.3.1. Software possesses the characteristic of modularity to the extent it is composed of a set of largely independent, individually cohesive, component parts. These parts are called modules. A module is defined as the smallest separately callable unit in any language.

A4.3.2. The decomposition of a software product into units and modules may occur at multiple levels. For example, an object-oriented development will contain classes that, in turn, contain data structures and callable functions. The evaluation of a multileveled decomposition for modularity is best performed at each level as appropriate. For example, the evaluation of source code questions for an Ada software product might best be performed at the package level of decomposition, the subprogram level of decomposition, or both.

A4.4. **Traceability.** Software possesses the characteristic of traceability when information can be traced between documents and between documents and source code. Software may be well written and well described, but still lack a clearly defined trail between top-level requirements and detailed implementation. The software maintainer must be able to trace any particular element from higher levels of program description down to executable code, and the reverse.

A4.5. **Consistency.**

A4.5.1. Software source code possesses the characteristic of consistency when software products correlate and contain uniform notation, terminology and symbology. In addition to uniformity, consistency requires the accurate representation of information. For example, a unit's description in the documentation should agree with (be consistent with) that description in the unit's source code.

A4.5.2. The use of certain conventions in I/O processing, error processing, unit interfacing, and naming of units and variables are typical reflections of consistency. Attention to consistency characteristics can aid in understanding the program. Consistency allows one to generalize easily. For example, programs using consistent conventions require that the format of units be similar. Thus, by learning the format of one unit (e.g., preface block format, declaration format, error checks) the format of all units is learned. This allows one to concentrate on understanding the true complexities of the processing.

A4.6. **Expandability.** Software possesses the characteristic of expandability when a physical change to information, computational functions, data storage, or execution time can be easily accomplished. Expandability is the ease with which physical changes can be made to the amounts and types of data a unit may process (sizing) and to the operations performed in a unit on its data (timing). Software may be perfectly understandable but not easily expandable. If the design of the program has not allowed for a flexible timing scheme or a reasonable storage margin, then even minor changes may be extremely difficult to implement.

C-1

STATEMENT: Identification data are included in a preface block.

CHARACTERISTIC: Descriptiveness

EXPLANATION: Each CSU should contain, within the preface block, general information about its development, including title, author, and revision history. CSU specifications and bodies should each have header blocks with this information.

GLOSSARY:
PREFACE BLOCK. Separate set of contiguous comments that usually precede the main body of the software component (a similar block at any other common software component location would be satisfactory) and that provide identification information, processing information, and interface information for the software component.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Figure C-1 shows an example of header identification data that shows the minimum amount of information you should expect to see.

```
-- ============================================================================
-- CSU TITLE:                Sensors
-- DESCRIPTION:              This CSU provides the temperature, humidity, and voltage sensor readings
--                           for sensors located in lab rooms. The information is provided to the Room
--                           Environment Control CSU in the SUCS CSCI.
-- CREATION DATE:            17 Sep 87
-- AUTHOR:                   B. Gates, MS Dev Group
-- REVISION HISTORY:
--                           3 Nov 88: Added check for divide by zero.
--                                     Al B. Kurky, NM Mx Team
--                           4 Nov 92: Removed unneeded check for divide by zero
--                                     S. Jobs, Apple Mx Team
-- FUNCTIONS AND PROCEDURES:
--                           function Thermometer_Reading
--                           function Hygrometer_Reading
--                           function Voltmeter_Reading
--                           procedure Put_In_Grid
-- GLOBALS:                  none
-- CONSTANTS:                none
-- ============================================================================
```

Figure C-1. Identification Data.

C-2

STATEMENT: Identifier names within this CSU (exclusive of identifiers in the modules but to include module names) are descriptive of their use.

CHARACTERISTIC: Descriptiveness

EXPLANATION: When modules and data items are named descriptively, the time it takes a maintainer to find a function and understand it is decreased. This involves more than the use of a naming convention because, by themselves, naming conventions do not necessarily ensure descriptiveness.

GLOSSARY:
DATA ITEM. Any parameter, variable, or constant.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Figure C-2 shows an Ada package specification that uses self-documenting identifiers.

```
package Sensors is

    type    ROOM_ID_TYPE is range 100..500;
    type    TEMPERATURE_TYPE is delta 0.1 range 0.0 .. 100.0;
    type    HUMIDITY_TYPE is delta 0.5 range 0.0 .. 100.0;
    type    VOLTAGE_TYPE is delta 0.01 range 0.0 .. 150.0;

    function Thermometer_Reading (Room:  in ROOM_ID_TYPE) return TEMPERATURE_TYPE;
    function Hygrometer_Reading(Room: in ROOM_ID_TYPE) return HUMIDITY_TYPE;
    function Voltmeter_Reading(Room: in ROOM_ID_TYPE) return VOLTAGE_TYPE;
    procedure Put_In_Grid(Room:  in ROOM_ID_TYPE; Temperature: in TEMPERATURE_TYPE;
                          Humidity: in HUMIDITY_TYPE;  Voltage: in VOLTAGE_TYPE);
end Sensors;
```

Figure C-2. Descriptive Identifier Names.

C-3

STATEMENT: The number of modules within this CSU is manageable.

CHARACTERISTIC: Simplicity

EXPLANATION: In general, a lot of complex modules in a CSU make it less understandable. Consider the number, size, and complexity of the modules in the CSU.

GLOSSARY: None

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: None

C-4

STATEMENT: Structured programming has been used in this CSU.

CHARACTERISTIC: Modularity

EXPLANATION: In most languages, nearly all processing is accomplished in modules rather than CSUs. For this reason, the evidence of structured programming at the CSU level is how well the CSU is organized into modules. Each module should perform a manageable portion of the CSU's overall functionality. Contrast this with the evaluation of structured programming at the module level which focused on control structures and processing flow.

GLOSSARY:
STRUCTURED PROGRAMMING. Any software development technique that includes structured design and results in the development of structured programs.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Figure C-4 shows a CSU that adheres to rules of a structured program. The CSU uses a top-down design approach to break out the lower level functions. An Ada package is used to declare the functions, define types, and define valid ranges for the types.

```
package Sensors is

    type    ROOM_ID_TYPE is range 100..500;
    type    TEMPERATURE_TYPE is delta 0.1 range 0.0 .. 100.0;
    type    HUMIDITY_TYPE is delta 0.5 range 0.0 .. 100.0;
    type    VOLTAGE_TYPE is delta 0.01 range 0.0 .. 150.0;

    function Thermometer_Reading (Room:  in ROOM_ID_TYPE) return TEMPERATURE_TYPE;
    function Hygrometer_Reading(Room: in ROOM_ID_TYPE) return HUMIDITY_TYPE;
    function Voltmeter_Reading(Room: in ROOM_ID_TYPE) return VOLTAGE_TYPE;
    procedure Put_In_Grid(Room:  in ROOM_ID_TYPE; Temperature: in TEMPERATURE_TYPE;
                          Humidity: in HUMIDITY_TYPE;  Voltage: in VOLTAGE_TYPE);
end Sensors;

package body Sensors is

    function Thermometer_Reading (Room:  in ROOM_ID_TYPE) return TEMPERATURE_TYPE is
    begin  -- processing begins here
    end Thermometer_Reading;

    function Hygrometer_Reading(Room: in ROOM_ID_TYPE) return HUMIDITY_TYPE is
    begin  --  processing begins here
    end Hygrometer_Reading;

    function Voltmeter_Reading(Room: in ROOM_ID_TYPE) return VOLTAGE_TYPE is
    begin  --  processing begins here
    end Voltmeter_Reading;

    procedure Put_In_Grid(Room:  in ROOM_ID_TYPE; Temperature: in TEMPERATURE_TYPE;
                          Humidity: in HUMIDITY_TYPE; Voltage: in VOLTAGE_TYPE) is
    begin  -- processing begins here
    end Put_In_Grid;
end Sensors;
```

Figure C-4. Structured Programming.

C-5

STATEMENT: This CSU exhibits loose coupling to other CSUs.

CHARACTERISTIC: Modularity

EXPLANATION: Coupling is a description of the strength of association between different CSUs. Coupling depends on the interface complexity between CSUs, the point which entry or reference is made to a CSU, and what data pass across the interface. High coupling is a negative characteristic of a system since an attempt to modify one CSU may result in the maintainer affecting other CSUs without being aware of the interaction between the CSUs.

GLOSSARY:
COUPLING. The manner and degree of interdependence between software components. Types include common-environmental coupling, data coupling, hybrid coupling, and pathological coupling.
INPUTS. (1) Pertaining to data received from an external source. (2) Pertaining to a device, process, or channel involved in receiving data from an external source. (3) To receive data from an external source. (includes values returned from function calls, parameters, or any global data) (4) To provide data from an external source.
OUTPUTS. (1) Pertaining to data transmitted to an external destination. (2) Pertaining to a device, process, or channel involved in transmitting data to an external destination. (3) To transmit data to an external destination.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Figure C-5(a) shows different types of coupling that can occur. Figure C-5(b) shows the varying degrees of coupling. This scale is nonlinear and data coupling (low coupling) is much better than control coupling, which is far better than content coupling (high coupling). Figure C-5(c) illustrates CSU coupling.

TYPE	LEVEL OF COUPLING
Data Coupling	One module serves as input to another module.
Stamp Coupling	When a portion of data structure, rather than simple arguments is passed via a module interface.
Control Coupling	One module communicates information to another module for the explicit purpose of influencing the second module's execution.
External Coupling	Modules are tied to an environment external to software, such as I/O couples to peripheral devices.
Common Environment Coupling	Two modules access a common data area.
Pathological Coupling	One software module affects or depends upon the internal implementation of another.
Content Coupling	All the contents of one module are included in the contents of another module.

Figure C-5(a). Types of Coupling.

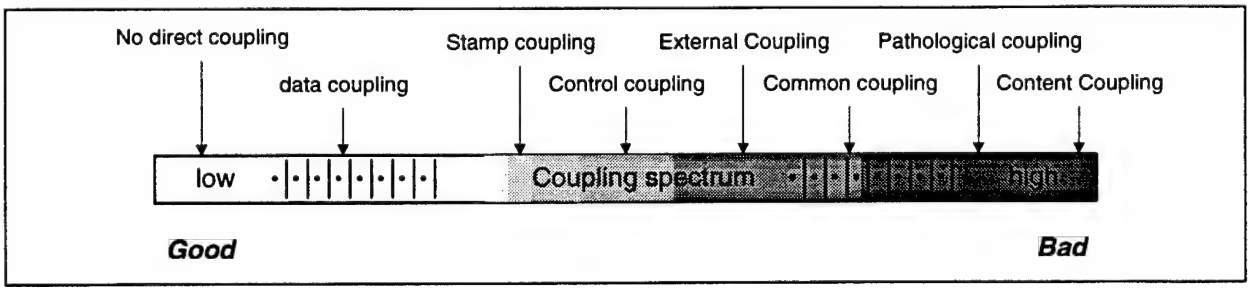

Figure C-5(b). Coupling Spectrum.

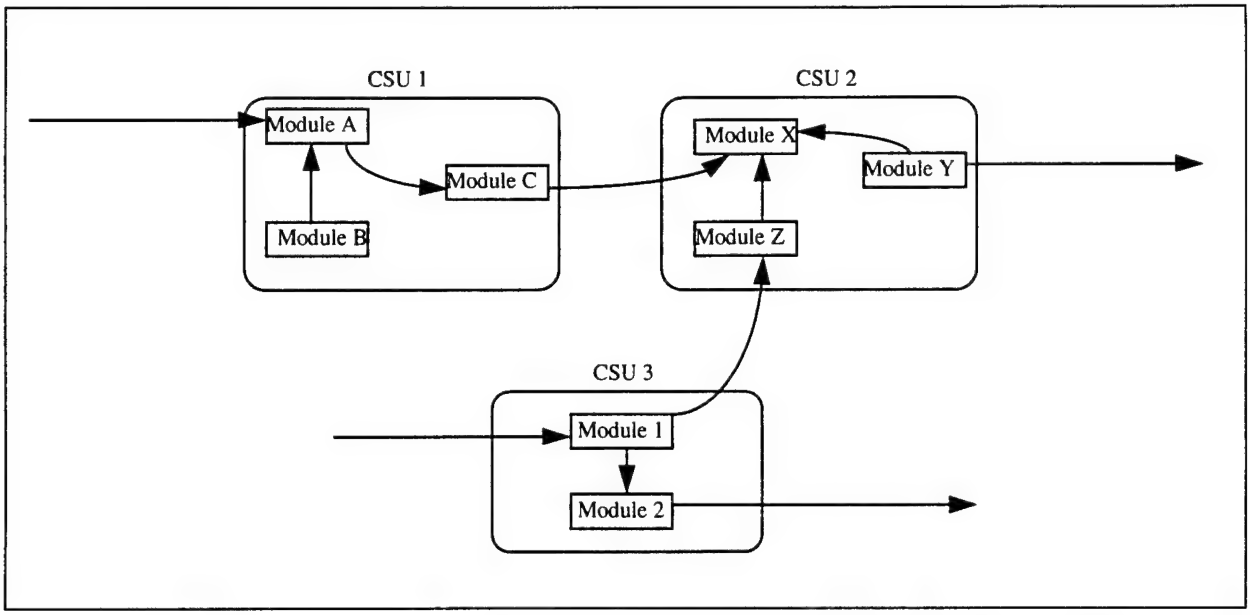

Figure C-5(c). Example of CSU Coupling.

C-6

STATEMENT: This CSU exhibits a high level of cohesion.

CHARACTERISTIC: Modularity

EXPLANATION: Cohesion describes the strength of association of the elements ***inside*** a CSU. All elements of CSU should be strongly associated (highly cohesive) and involved in performing the same task. Consider relationships between the modules of the CSU. Ideally, a highly cohesive CSU should contain a set of strongly related functions that perform a single task.

GLOSSARY:
COHESION. The manner and degree to which the tasks performed by a single software component are related to one another. Types include coincidental, communicational, functional, logical, procedural, sequential, and temporal.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Figure C-6(a) shows different types of cohesion. Figure C-6(b) shows a cohesion spectrum from "bad" to "good." The scale is nonlinear and low-end cohesiveness is much worse than middle range, which is nearly as "good" as high-end cohesion.

TYPE	LEVEL OF COHESION
Coincidental	Tasks performed by a module have no functional relationship to one another.
Logical	Tasks performed by a module perform logically similar functions; for example, processing of different types of input data.
Temporal	The tasks performed by a software module are all required at a particular phase of program execution; for example, a module containing all of a program's initialization tasks.
Procedural	Tasks performed by a module all contribute to a given program, such as an iteration or decision process.
Communicational	Tasks performed by a module use the same input data or contribute to producing the same output data.
Sequential	The output of one task performed by a module serves as input to another task performed within the module.
Functional	Tasks performed by a module all contribute to the performance of a single function.

Figure C-6 (a). Types of Cohesion.

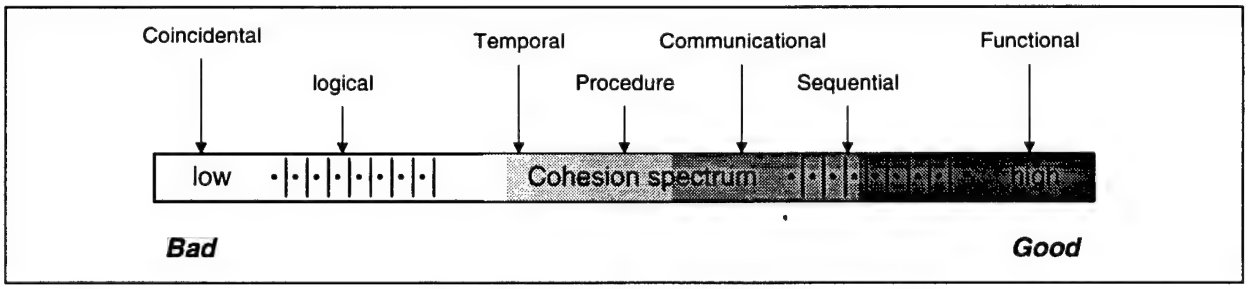

Figure C-6(b). Cohesion Spectrum.

C-7

STATEMENT: It is easy to trace from the CSU's source listing to its detailed description in the documentation.

CHARACTERISTIC: Traceability

EXPLANATION: All information about this CSU's source listing must also be easily found and located together in the documentation to facilitate changes. When looking for source code descriptions, the maintainer should be able to search the documentation, traceability matrix, on-line index, or other resource and find the desired information.

GLOSSARY: None

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: None

C-8

STATEMENT: CSU-level data items can be traced to their description in the documentation.

CHARACTERISTIC: Traceability

EXPLANATION: It is useful for the maintainer to have a link from the data used in the CSU to the detailed description of data in the documentation. Global data and parameters should be described in a data dictionary resource. Local data should be described within the source listings, the documentation describing the CSU, and may also appear in the data dictionary.

GLOSSARY:
DATA ITEM. Any parameter, variable, or constant.

SPECIAL RESPONSE INSTRUCTIONS: Answer N/A if there are no data items at the CSU level.

EXAMPLE: SET/USED listings, relationship matrix, data dictionary, etc.

C-9

STATEMENT: The information in the preface block of this CSU is consistent with the associated source code.

CHARACTERISTIC: Consistency

EXPLANATION: Preface block data can be extremely helpful to the maintainer in understanding the code, but to be helpful it must be correct (consistent with the source code) and complete (containing all required information).

GLOSSARY:
PREFACE BLOCK. Separate set of contiguous comments that usually precede the main body of the software component (a similar block at any other common software component location would be satisfactory) and that provide identification information, processing information, and interface information for the software component.

SPECIAL RESPONSE INSTRUCTIONS: Answer 1 if there is no preface block.

EXAMPLE: (Not Exhaustive)

- Inputs explained agree with inputs as coded.
- Outputs explained agree with outputs as coded.
- Calls explained agree with calls as coded.
- Processing information matches the processing within the unit.
- Error handling explained agree with the error handling within the code.

C-10

STATEMENT: The inputs and outputs to this CSU, as described in the documentation, correspond to the CSU's source listing.

CHARACTERISTIC: Consistency

EXPLANATION: It is extremely important for each CSU and its documentation to be in agreement with each other. Most CSUs interface with other CSUs, but the maintainer doing interface work usually views other CSUs mainly through the documentation. This question specifically addresses agreement between input and output parameters and related documentation.

GLOSSARY:
INPUTS. (1) Pertaining to data received from an external source. (2) Pertaining to a device, process, or channel involved in receiving data from an external source. (3) To receive data from an external source. (includes values returned from function calls, parameters, or any global data) (4) To provide data from an external source.
OUTPUTS. (1) Pertaining to data transmitted to an external destination. (2) Pertaining to a device, process, or channel involved in transmitting data to an external destination. (3) To transmit data to an external destination.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Figure C-10 illustrates inputs and outputs from a CSU. Module inputs and outputs are evaluated in the module questionnaire but are shown here for completeness.

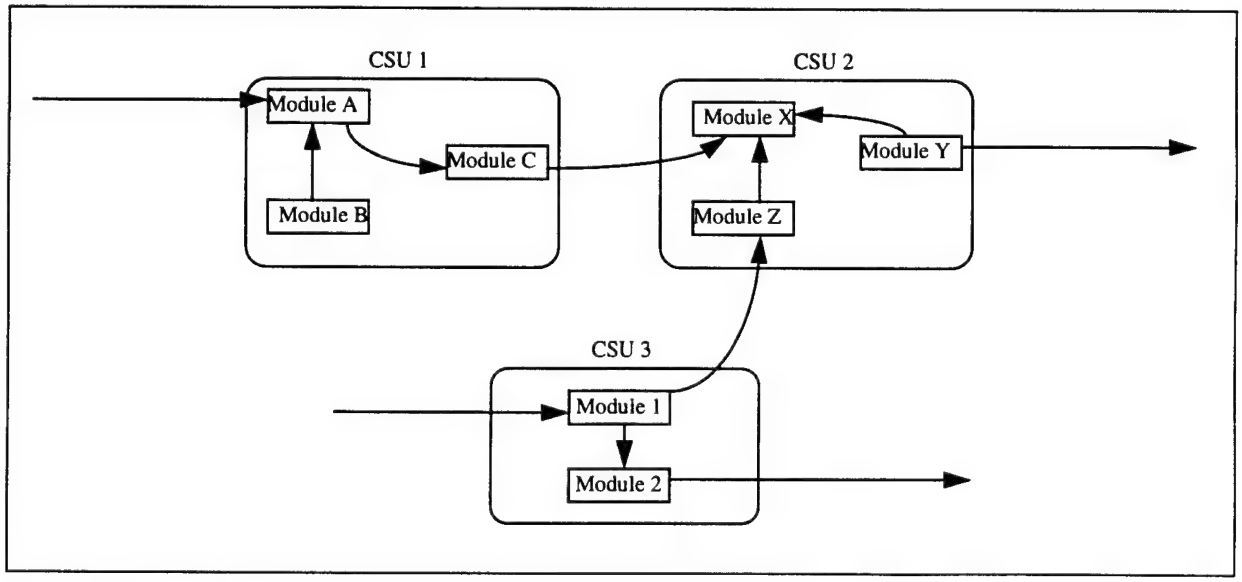

Figure C-10. Inputs and Outputs to/from a CSU.

C-11

STATEMENT: CSU-level constants and data structure dimensions are appropriately parameterized.

CHARACTERISTIC: Expandability

EXPLANATION: By parameterizing constants and data structure dimensions, any occurrence may be modified by simply changing the parameter. In contrast, if a value is given for the constant or dimension, every occurrence of the value must be found and modified whenever it is changed. If appropriate parameterization doesn't exist, answer based on the impact to the expandability of the module.

GLOSSARY:
DATA STRUCTURE. A physical or logical relationship among data elements, designed to support specific data manipulation functions.
PARAMETERIZED. Referenced by name, not by an actual value (e.g., PI instead of 3.1415926).

SPECIAL RESPONSE INSTRUCTIONS: Answer N/A if there are no constants or data structures at the CSU level.

EXAMPLE: Figure C-11 shows constants and dimensions that are parameterized. By parameterizing constants and data structure dimensions, any occurrence may be modified by simply changing the parameter. In contrast, if a value is given for the constant or dimension, every occurrence of the value must be found and modified whenever it is changed.

```
package Types is

--Preface block

    numb_cards    : CONSTANT Integer  := 52;

    type  SUIT_TYPE is (Clubs, Diamonds, Hearts, Spades);

    Jack          : CONSTANT Integer := 11;
    Queen         : CONSTANT Integer := 12;
    King          : CONSTANT Integer := 13;
    Ace           : CONSTANT Integer := 14;

    type CARD_SET_TYPE is ARRAY (2..Ace,Suit_Type) OF Boolean;

end Types;
```

Figure C-11. Parameterized Dimensions at the CSU Level.

C-12

STATEMENT: Overall, it appears that the characteristics of this CSU's source code listing contributes to the maintainability of the system.

CHARACTERISTIC: General

EXPLANATION: Consider all aspects of maintainability including any other aspects that were not covered in this questionnaire. In addition to a 6 through 1 answer, provide any comments that reflect the overall maintainability of the CSU.

GLOSSARY: None

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: None

SOFTWARE IMPLEMENTATION QUESTIONS

A5.1. The questions listed within this section are used to evaluate those aspects of the software product that can only be fully evaluated by examining both the documentation and source code. It should be noted that some of the questions are similar to questions in the source code questionnaire; however, the questions in the implementation questionnaire pertain to the entire system, not individual modules.

A5.2. The implementation characteristics evaluated are Convention, Simplicity, Modularity, and Expandability (see figure A5-1) Each question in the questionnaire is used to evaluate an aspect of one of these characteristics.

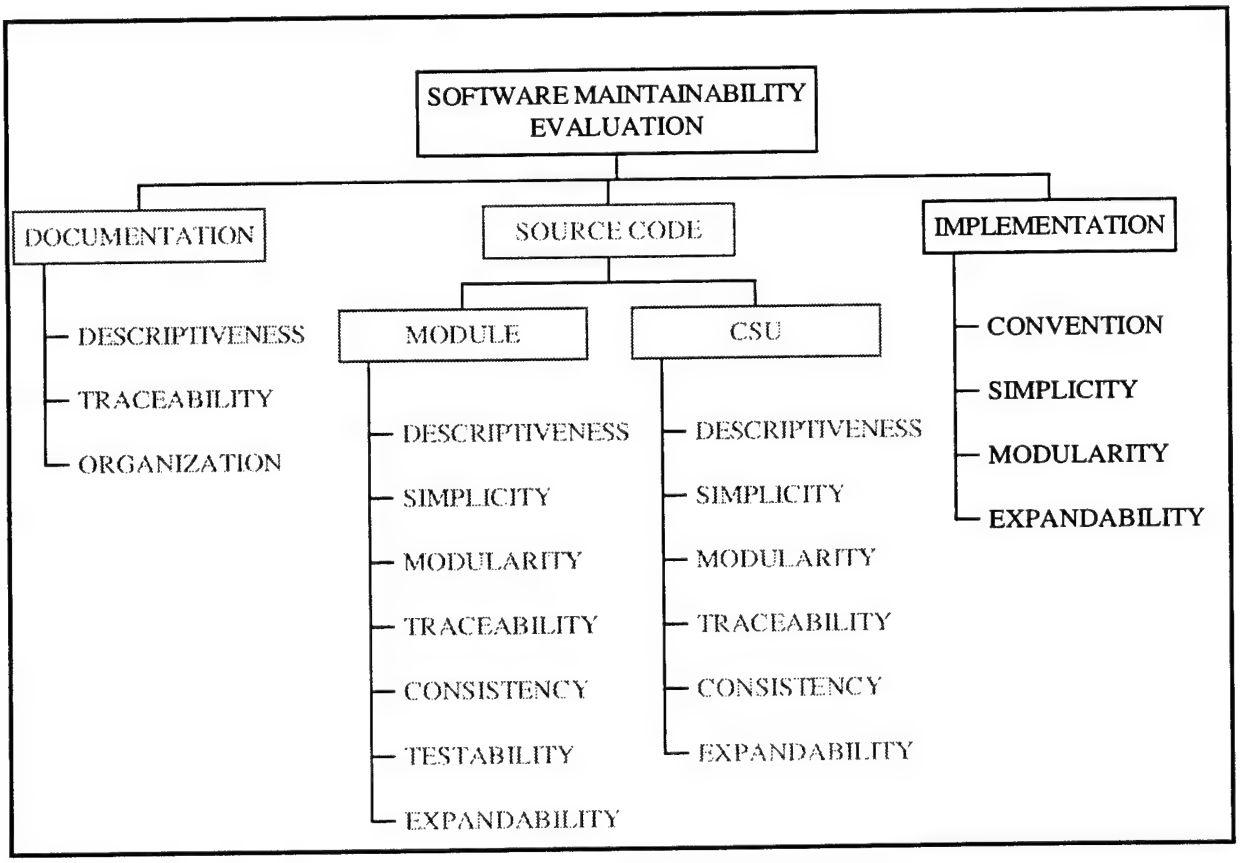

Figure A5-1. Software Implementation Characteristics.

A5.3. Convention.

A5.3.1. The use of good engineering conventions (standards) aids in understanding software. The existence and consistent use of conventions results in software products that are well organized and contain uniform notation, terminology, and symbology.

A5.3.2. Conventions allow one to generalize easily. For example, software developers should follow a standard convention for the format of preface block information in modules. Consequently, by learning the format of one module preface block, the format of all modules is learned. This allows one to concentrate on understanding the information provided, rather than finding it.

A5.4. Simplicity.

A5.4.1. Simple source code is easier for programmers to understand. The implementation of a software system is made simpler by using a high-order language (HOL), logically organizing the control and data flow, and using straightforward coding techniques.

A5.4.2. Simplicity is, in some respects, the essence of maintainability. The degree of simplicity exhibited by a software product is related to the understandability of the system. Maintainability is enhanced if there are fewer things the maintainer must understand or discriminate among and greater use of basic, commonly used techniques and structures.

A5.5. Modularity.

A5.5.1. Modular software is composed of individual functions that are isolated into low-level modules whose properties are easily provable. These low-level modules can then be used by more complex routines, which do not have to concern themselves with the details of the low-level modules, only their function. The complex routines may themselves be viewed as modules by still-higher-level routines, which use them independently of their internal details.

A5.5.2. Aggregating the set of implementation-dependent trouble spots into small, easily identifiable units is an important method of making a program easier to maintain. When properly done, this functional decomposition makes the system easier to understand and makes modifications to the software more localized and straightforward.

A5.5.3. The decomposition of a software product into component parts occurs at multiple levels. For example, an object-oriented development will contain classes that, in turn, contain data structures and callable functions. The evaluation of modularity should consider each level of decomposition as appropriate.

A5.6. Expandability. A software component is expandable when sufficient memory and processor power exits to allow for growth in the both the software size (executable lines of code) and the amount of data processed.

I-1

STATEMENT: A useful naming convention for modules and data items has been followed.

CHARACTERISTIC: Convention

EXPLANATION: Useful naming conventions help the programmer by:

- describing the purpose of modules and the meaning of data items.
- improving the readability and understanding of processing.
- helping to differentiate between local and global data.
- aiding in the recognition of external function calls.

This list is not intended to be exhaustive. There are many more reasons for naming conventions. Since naming conventions cannot always be determined from a sample of source code, consider looking in the software development plan or other documentation for programming standards or conventions.

GLOSSARY:
GLOBAL DATA ITEM. Data that can be accessed by two or more nonnested modules of a computer program without being explicitly passed as parameters between the modules.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE:

- All global variables must begin with "XG" (external-global); no other type of variable name begins with that letter combination.
- "K" must be the first letter in all constants.
- Modules contained within one CSCI of a multi-CSCI system are identified by the CSCI in which they are contained. For instance: *SM_signal_hdlr*, *SM_key_handler*, *SM_getkey*, and *SM_get_char* are all modules that fall under the Screen Management (SM) CSCI.
- Module names should describe their purpose. For example: *FC_calculate_target_solution* is more understandable than *FC_Step3* for a module in the Flight Control (FC) CSCI of an aircraft system.
- Underscores should be used between words in module and data item names to improve readability. For instance, the global constant item *K_max_target_array_size* is easier to read than *Kmaxtargetarraysize*.

I-2

STATEMENT: A useful convention for comments has been followed.

CHARACTERISTIC: Convention

EXPLANATION: Conventions should provide a template for source code format. This format should allow the programmer to easily differentiate comments from the executable code. The placement, use of, and delineation of comments which clarify the purpose of the code should be uniform throughout the program. Comments should not be delineated in a variety of ways (e.g., some are bordered by ampersands and some by asterisks, or some comments are in-line while others are blocked off).

GLOSSARY: None

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: None

I-3

STATEMENT: Useful programming conventions for interfacing modules have been followed.

CHARACTERISTIC: Convention

EXPLANATION: Module interface design is extremely important; improper interfacing can lead to many hidden errors. Program design conventions should be documented. In addition, the module descriptions can be scanned to determine whether such conventions have been established and/or followed. Establishing linkage conventions is especially important for assembly language modules.

Inputs and outputs, both argument type and global data type, require coordination between the senders and the receivers. Such coordination requires explicit description of all attributes of each variable and should be listed in an interface control document.

GLOSSARY:
CONVENTIONS. Requirements employed to prescribe a disciplined uniform approach to providing consistency in a software product, that is, uniform patterns or forms for arranging data.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Look for multiple return statements, global variable usage (and how many globals are accessed from COMMON blocks or COMPOOLs), and parameter declarations (are they declared "call by reference" (like Ada's in out mode), "call by value" (like Ada's in mode), or "call by result" (like Ada's out mode).

In Ada, program units generally have a similar two-part structure consisting of a specification and a body. The specification identifies the information visible to the user of that program unit (the interface), while the body contains the unit implementation details which can be hidden from the user. The specification can be viewed as the interface design between modules.

I-4

STATEMENT: Useful programming conventions for I/O processing have been followed.

CHARACTERISTIC: Convention

EXPLANATION: Program I/O processing is the interface of the program to the rest of its external environment. The module descriptions should be skimmed to determine whether any particular design consistency/conventions have been followed for program I/O processing.

One module or set of modules should be clearly identified as interfacing the system to the environment. All attributes of all inputs and all outputs should be clearly identified. These data are essential to all personnel interfacing with any I/O data, whether externally (to/from real world) or internally (to/from processing routines).

GLOSSARY:
CONVENTIONS. Requirements employed to prescribe a disciplined uniform approach to providing consistency in a software product, that is, uniform patterns or forms for arranging data.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Some conventions may be:

- Input buffer flush after each read.
- Output buffer flush before every write.
- Input data checks.
- Ending each data file with a ^Z.
- I/O done only by one set of modules.
- Leaving the cursor in the same place after screen I/O.
- The order of the bytes sent over a communication line (i.e., least significant first or most significant first).

I-5

STATEMENT: Useful error processing conventions have been followed.

CHARACTERISTIC: Convention

EXPLANATION: Error processing conventions could specify local or centralized handling of errors. It is simpler for the maintainer to understand if the entire CSCI follows a consistent approach.

Under a centralized error processing approach, any module that communicates an error condition to an error processing routine must do so properly. Therefore, error processing procedures (conventions) must be documented and followed.

GLOSSARY:
ERROR PROCESSING. The steps required to set program data and control states following the detection of an undesirable event.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: An error type is generated and passed to the error processing routines. The routine generating the error type "knows" that the error processing routine will handle it properly when both parties have followed the documented procedures. Some conventions may include:

- Having special error processing modules that handle all error processing.
- All modules that encounter an error would perform the same action.
- Writing out error messages in a standard format.
- Every module has error checking, instead of just having some programmers putting in error checking and other programmers ignoring the issue.

I-6

STATEMENT: Useful standards for design and source code development have been followed.

CHARACTERISTIC: Convention

EXPLANATION: Adherence to design and coding standards ensures that the coding will be more readily understandable by anyone involved in the project. These standards should be documented, and are typically found or referenced in the software development plan (SDP).

GLOSSARY: None

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: (Not Exhaustive)

- Modules of 100 lines of code or less.
- Modules start on a new page.
- PDL will be used to describe processing.

I-7

STATEMENT: The source code is implemented in a high-order language (HOL).

CHARACTERISTIC: Simplicity

EXPLANATION: High-order languages have many advantages over low-level languages. Most HOLs are (1) easier to understand and trace, (2) facilitate structured programming, and (3) increase portability by reducing links to a specific computer architecture. If a non-DoD approved HOL is used, verify that a waiver exists. Ada, in particular, has been mandated for DoD use, in part because of its maintainability benefits.

In addition to *what* language was used, consider *how* the language was used. Appropriate capabilities of a language should be used (e.g., CASE constructs should be used instead of multiple nested IF statements). Even assembly languages can be structured through use of macros.

Some applications require low-level languages because no high-order language compiler exists for the hardware. While this situation may be unavoidable, the result is software that is more difficult to maintain and which requires more maintainer knowledge of hardware operating characteristics.

GLOSSARY:
HIGH-ORDER LANGUAGE (HOL). A programming language that requires little knowledge of the computer on which it will run, can be translated into several different machine languages, allows symbolic naming, and usually results in several machine instructions for each program statement.
LOW-LEVEL LANGUAGE. A programming language that corresponds closely to the instruction set of a given computer.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: None

I-8

STATEMENT: Control flow within this CSCI is logically organized.

CHARACTERISTIC: Simplicity

EXPLANATION: Systems become complex when they consist of too many or too few levels within their hierarchy.

The documentation should include an overview section which will describe the overall control flow in narrative or chart form. Evaluators should determine how easily they can follow control flow. You may also be able to determine control flow from a module called/calling listing, but this would be a tedious task. Any program should have some sort of module structure chart that shows how modules interact during execution.

GLOSSARY:
CONTROL FLOW. The sequence in which operations are performed during the execution of a computer program.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: None

I-9

STATEMENT: Data flow within this CSCI is logically organized.

CHARACTERISTIC: Simplicity

EXPLANATION: Systems become complex when they consist of too many or too few levels within their hierarchy. Symptoms of poor data flow include excessive fan-in and fan-out. Too few levels of hierarchical structure result in excessive fan-out. Too many levels of structure result in excessive numbers of relationships between modules.

The documentation should include an overview section which will describe the overall data flow in a data flow diagram or narrative form. Evaluators should determine how easily they can follow the flow of data. You may also be able to determine data flow from a data dictionary or module called/calling listing, but this would be a tedious task. Any program should have some sort of data flow diagram that shows where data comes from, how they transformed, and where they go.

GLOSSARY:
DATA FLOW. The sequence in which data transfer, use and transformation are performed during the execution of a computer program.
FAN-IN. A measure of the number of software components that directly control a given software component.
FAN-OUT. A measure of the number of software components that are directly controlled by a given software component.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: None

I-10

STATEMENT: A knowledge of mathematics beyond basic algebra is not required to understand the mathematical functions performed by the program.

CHARACTERISTIC: Simplicity

EXPLANATION: In general, complex mathematical algorithms require advanced mathematics knowledge, system domain knowledge, and programming experience. When used, maintainers will require more education and training. Modifications to these algorithms will also require more resources.

GLOSSARY:
BASIC ALGEBRA. Functions (including trigonometric and geometric functions), equations, polynomials, graphing of functions, basic manipulations, Booleans, etc. This specifically excludes calculus, differential equations, Fourier transforms, statistical techniques, and control theory.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: None

I-11

STATEMENT: Resource allocation is transparent to the programmer.

CHARACTERISTIC: Simplicity

EXPLANATION: Systems that use dynamic allocation become more complex if they are required to perform their own resource management rather than relying on an operating system.

The sharing or dynamic reassignment of resources should be a highlight of a section describing special processing (control) considerations, memory allocation, timing requirements by mission phase, etc. The evaluator can check the individual module descriptions for possible mention of any dynamic resource allocation and check documentation for information about whether these tasks are performed by the CSCI or the operating system.

GLOSSARY:
RESOURCE ALLOCATION. The assignment of computer resources to current and waiting jobs; for example, the assignment of main memory, input/output devices, and auxiliary storage to jobs executing concurrently in a computer system.

SPECIAL RESPONSE INSTRUCTIONS: Answer 6 if resource allocation is clearly fixed throughout or is controlled by the operating system.

EXAMPLE: (Not Exhaustive)

- Linked lists
- Stacks
- Binary trees
- Paging
- Pointer operations
- File I/O
- Operating system calls (even some embedded systems have these)

I-12

STATEMENT: The use of shared memory is not excessive.

CHARACTERISTIC: Simplicity

EXPLANATION: Use of shared memory makes source code more difficult to maintain. When it is used, it should be clearly identified in source listings and documentation.

GLOSSARY:
SHARED MEMORY. Memory locations that can be referenced by two or more different variable names.

SPECIAL RESPONSE INSTRUCTIONS: Answer 6 if shared memory is not used.

EXAMPLE: Memory sharing techniques would include FORTRAN's EQUIVALENCE operation, JOVIAL's OVERLAY or POS operation, and memory overlay techniques.

I-13

STATEMENT: The use of recursive or reentrant programming is not excessive.

CHARACTERISTIC: Simplicity

EXPLANATION: The documentation should specify within a general "program design considerations" section or the individual module description section whether recursion or reentrancy is to be utilized. Many languages (or at least a particular implementation of a compiler) do not allow recursive or reentrant code. Some languages (e.g., stack-oriented languages like JOVIAL and Ada) allow recursion as a natural language capability.

GLOSSARY:
RECURSIVE PROGRAMMING. A process in which a software module calls itself.
REENTRANT PROGRAMMING. Pertaining to a software module that can be entered as part of one process while also in execution as part of another process and still achieve the desired results.

SPECIAL RESPONSE INSTRUCTIONS: Answer 6 if there are no occurrences of recursive or reentrant code within the CSCI.

EXAMPLE: Figure I-13(a) shows an Ada example to compute the factorial (n!) of a number. using recursion. Figure I-13(b) uses simple iteration, which uses a more common and straightforward coding technique and is generally considered to be easier to understand and maintain.

(a)

```
FUNCTION Factorial (N : IN Natural) RETURN
Positive IS
    -- Computes the factorial of N (N!) recursively
    BEGIN
        IF N = 0 THEN
            RETURN 1;
    --Stopping Case
        ELSE  -- recursion
            RETURN N * Factorial(N-1);
        END IF;
    END Factorial;
```

(b)

```
FUNCTION Factorial (N : IN Natural) RETURN
Positive IS
    -- Computes the factorial of N (N!) iteratively
        Result : Positive;
    BEGIN
        Result := 1;
        FOR Count IN 2 . . N LOOP
            Result := Result * Count;
        END LOOP;
        RETURN Result;
    END Factorial;
```

Figure I-13. Recursion.

I-14

STATEMENT: CSCI initialization is done by one (set of) modules designed exclusively for that purpose.

CHARACTERISTIC: Modularity

EXPLANATION: It is usually better if each function or module does not handle its own global initialization. There should be one component (e.g., a set of modules) that performs general program initialization. Every program does some initialization, even embedded systems. This question is directed at how well the modules that perform this function are partitioned.

The documentation describing the functions and control flow should also describe how the initial program state is determined (e.g., via execution of one initialization module or not). Checks of module processing may indicate whether any initialization processing is mixed with other application functions.

If the partitioning is such that each function is performed by a set of modules and there is one module in each set expressly for initialization, then strong agreement with the question should be indicated. In addition, if these initialization modules are all executed in preparation for any other functional activity as the first program action, then there should be essentially complete agreement with this question's statement.

GLOSSARY:
EMBEDDED SYSTEM. A computer system that is part of a larger system and performs some of the requirements of that system; for example, a computer system used in an aircraft or rapid transit system.
INITIALIZATION. The preparatory steps required to set the initial program data and control states.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: None

I-15

STATEMENT: The CSCI's external interface is implemented by one (set of) modules designed exclusively for that purpose.

CHARACTERISTIC: Modularity

EXPLANATION: It is usually better if each function or module does not handle its own I/O. There should be one component (e.g., a few modules) of the program which performs input processing, and one which performs output processing. This isolates the internal processing logic from changes to the external format of inputs and outputs.

The documentation describing the program functions and control flow should also describe how the program I/O is done (e.g., via execution of one or more modules). Checks of module processing may indicate whether any I/O functions are mixed with other application functions.

If the partitioning is such that each function is performed by a set of modules and there is one module in each set expressly for I/O processing, then strong agreement with the question should be indicated.

GLOSSARY:
EXTERNAL INTERFACE. A boundary at which independent computer programs interact or at which a computer program interacts with hardware (e.g., program input/output data or interrupts) or users (e.g., human-computer interface).

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Figure I-15 shows an Ada package that declares a set of modules that act as a external interface.

```
package Sensors is

    type    ROOM_ID_TYPE is range 100..500;
    type    TEMPERATURE_TYPE is delta 0.1 range 0.0 .. 100.0;
    type    HUMIDITY_TYPE is delta 0.5 range 0.0 .. 100.0;
    type    VOLTAGE_TYPE is delta 0.01 range 0.0 .. 150.0;

    function Thermometer_Reading (Room:  in ROOM_ID_TYPE) return TEMPERATURE_TYPE;
    function Hygrometer_Reading(Room: in ROOM_ID_TYPE) return HUMIDITY_TYPE;
    function Voltmeter_Reading(Room: in ROOM_ID_TYPE) return VOLTAGE_TYPE;
    procedure Put_In_Grid(Room:  in ROOM_ID_TYPE; Temperature: in TEMPERATURE_TYPE;
                          Humidity: in HUMIDITY_TYPE;  Voltage: in VOLTAGE_TYPE);
end Sensors;
```

Figure I-15. External Interface Modules.

I-16

STATEMENT: CSCI termination is done by one (set of) modules designed exclusively for that purpose.

CHARACTERISTIC: Modularity

EXPLANATION: It is usually better if each function or module does not handle its own termination. There should be one component (e.g., a few modules) of the program which is for termination processing.

The documentation describing the program functions and control flow should also describe how the final program state is determined (e.g., via execution of one termination module or not). Checks of module processing may indicate whether any termination processing (e.g., FORTRAN's STOP statement) is mixed with other application functions.

If the partitioning is such that each function is performed by a set of modules and there is one module in each set expressly for termination processing, then strong agreement with the question should be indicated. If, in addition, these termination modules are executed only as a systematic program termination procedure, then there should be essentially complete agreement with this question's statement.

GLOSSARY:
TERMINATION. The steps required to set the final program data and control states (due to normal or abnormal termination).

SPECIAL RESPONSE INSTRUCTIONS: Answer N/A if no termination steps are required (e.g., bomb or missile).

EXAMPLE: For some systems the normal mode of termination may be abrupt and unpredictable (i.e., turning off the power or for a bomb or missile, detonation). But most embedded applications still need to perform some type of processing before being turned off. Some examples include:

- Declassifying classified memory areas.
- Writing maintenance data out to a Data Transfer Medium.
- Writing operations data (postflight debrief) information out to a Data Transfer Medium.

I-17

STATEMENT: Loose coupling has been implemented in this CSCI.

CHARACTERISTIC: Modularity

EXPLANATION: Coupling is a description of the strength of association between different software components. Coupling depends on the interface complexity between components, the point which entry or reference is made to a component, and what data passes across the interface. High coupling is a negative characteristic of a system since an attempt to modify one component may result in the maintainer affecting other components without being aware of the interaction between the them.

GLOSSARY:
COUPLING. The manner and degree of interdependence between software components. Types include common-environmental coupling, data coupling, hybrid coupling, and pathological coupling.
INPUTS. (1) Pertaining to data received from an external source. (2) Pertaining to a device, process, or channel involved in receiving data from an external source. (3) To receive data from an external source. (includes values returned from function calls, parameters, or any global data) (4) To provide data from an external source.
OUTPUTS. (1) Pertaining to data transmitted to an external destination. (2) Pertaining to a device, process, or channel involved in transmitting data to an external destination. (3) To transmit data to an external destination.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Figure I-17(a) shows different types of coupling that can occur. Figure I-17(b) shows the varying degrees of coupling. This scale is nonlinear and data coupling (low coupling) is much better than control coupling, which is far better than content coupling (high coupling).

TYPE	LEVEL OF COUPLING
Data Coupling	One module serves as input to another module.
Stamp Coupling	When a portion of data structure rather than simple arguments is passed via a module interface.
Control Coupling	One module communicates information to another module for the explicit purpose of influencing the second module's execution.
External Coupling	Modules are tied to an environment external to software, such as I/O couples to peripheral devices.
Common Environment Coupling	Two modules access a common data area.
Pathological Coupling	One software module affects or depends upon the internal implementation of another.
Content Coupling	All the contents of one module are included in the contents of another module.

Figure I-17(a). Types of Coupling.

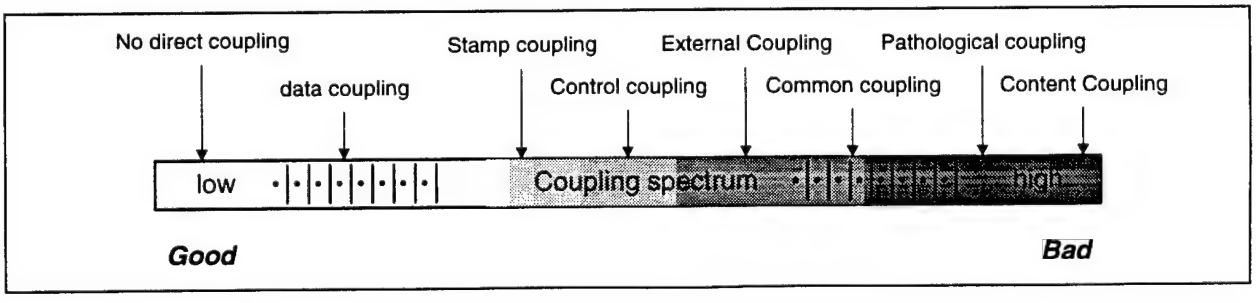

Figure I-17(b). Coupling Spectrum.

I-18

STATEMENT: This CSCI exhibits a high level of cohesion.

CHARACTERISTIC: Modularity

EXPLANATION: Cohesion describes the strength of association of the elements ***inside*** a component. All elements of the CSCI should be strongly associated (highly cohesive) and involved in performing the same task. A cohesive CSCI performs a single major task, requiring little interaction with other CSCIs in the program.

GLOSSARY:
COHESION. The manner and degree to which the tasks performed by a single software component are related to one another. Types include coincidental, communicational, functional, logical, procedural, sequential, and temporal.

SPECIAL RESPONSE INSTRUCTIONS: None

EXAMPLE: Figure I-18(a) shows different types of cohesion. Figure I-18(b) shows a cohesion spectrum from "bad" to "good." The scale is nonlinear and low-end cohesiveness is much worse than middle range, which is nearly as "good" as high-end cohesion.

TYPE	LEVEL OF COHESION
Coincidental	Tasks performed by a module have no functional relationship to one another.
Logical	Tasks performed by a module perform logically similar functions; for example, processing of different types of input data.
Temporal	The tasks performed by a software module are all required at a particular phase of program execution; for example, a module containing all of a program's initialization tasks.
Procedural	Tasks performed by a module all contribute to a given program, such as an iteration or decision process.
Communicational	Tasks performed by a module use the same input data or contribute to producing the same output data.
Sequential	The output of one task performed by a module serves as input to another task performed within the module.
Functional	Tasks performed by a module all contribute to the performance of a single function.

Figure I-18(a). Types of Cohesion.

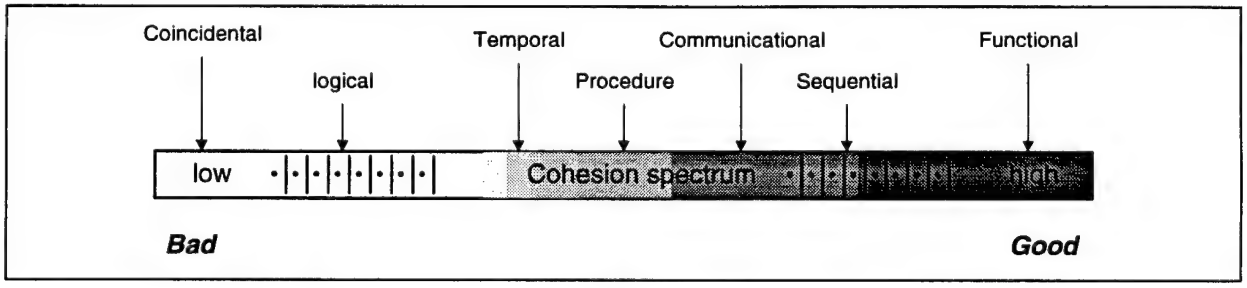

Figure I-18(b). Cohesion Spectrum.

I-19

STATEMENT: Global data items are partitioned into logically related sets.

CHARACTERISTIC: Modularity

EXPLANATION: Most computer languages provide mechanisms for grouping data items. Examples are common blocks, structures, arrays, and packages. Modularity is enhanced if global data are partitioned into groups according to logical relationships. The documentation should include descriptions of how the global data are partitioned. These structures should be examined to determine the extent to which their elements are related.

GLOSSARY:
GLOBAL DATA ITEM. Data that can be accessed by two or more nonnested modules of a computer program without being explicitly passed as parameters between the modules.

SPECIAL RESPONSE INSTRUCTIONS: Answer 6 if there are no global data items.

EXAMPLE: Figure I-19 shows a method of grouping data in Ada.

```
type Sector is
      record
            radius : length;
            width : angle;
      end record;

type radar_track is
      record
            here : position;
            now : time_of_day;
            region: Sector
      end record;
```

Figure I-19. Logical Grouping of Data Items.

I-20

STATEMENT: Information hiding has been used in this CSCI.

CHARACTERISTIC: Modularity

EXPLANATION: To control software complexity, it often useful to limit the amount of information that each module can "see." Each module "owns" certain data objects on which it operates. The ability to operate on an object implies an understanding of the object's internal structure. A module should give other modules enough information to properly declare common objects, but not so much information that the other modules can also operate on the objects. Information hiding makes is easier to modify a data structure because only one module is dependent on the internal organization of the object.

GLOSSARY:
ENCAPSULATION. A software development technique that consists of isolating a system function or a set of data and operations on those data within a module and providing precise specifications for the module.
INFORMATION HIDING. A software development technique in which each module's interfaces reveal as little as possible about the module's inner workings and other modules are prevented from using information about the module that is not in the module's interface specification.

SPECIAL RESPONSE INSTRUCTIONS: In addition to a 6 through 1 answer, provide comments on the overall maintainability of the implementation of the CSCI. If there was an aspect of the implementation that was not covered, but is relevant, provide specific comments.

EXAMPLE: Figure I-20: shows a method to construct private types in C. The private types are generated by conditionally compiling two declarations of the symbol data structure in *sym.h*. The detailed declaration is compiled for the *sym.c* module, while a deliberately vague declaration is compiled for the other modules.

```
    //file name:          sym.h
struct    _private_type_sym
{
    char    *sym_name;
    short   sym_value;
};

#if SYM_OWNER
    typedef struct _private_type_sym SYM;
#else
    typedef struct {
        char _x[sizeof (struct _private_type_sym)];
    } SYM;
```

Figure I-20. C Information Hiding.

I-21

STATEMENT: The impact of changes to sizing can be easily determined.

CHARACTERISTIC: Expandability

EXPLANATION: Since modifications to the system may result in additional consumption of memory and/or secondary storage, maintainers must be able to easily determine the sizing impacts of changes. Maintenance personnel should have procedures/tools available to inform them of any memory reserve impacts to the system resulting from software changes.

GLOSSARY: None

SPECIAL RESPONSE INSTRUCTIONS: Answer N/A if there are clearly no sizing constraints in this CSCI.

EXAMPLE: Figure I-21 shows a table for a flight control computer which might be used to determine the sizing impacts of changes after the fact. Maintainers should be able to get this information easily and know what the reserve limits are.

	PAGE	MAXIMUM	CURRENT	AVAILABLE	% USAGE
Instruction Memory	0	57,344	27,648	29,696	48.00
	1	57,344	0	57,344	0.00
	2	8,192	0	8,192	0.00
	3	8,192	0	8,192	0.00
	Total	131,072	27,648	103,424	21.09
Data Memory	All	28,672	16,384	12,288	57.14
Scratchpad Memory	All	8,192	4,332	3,860	52.88
PFCS Dual Port RAM	All	4,096	3,036	1,060	74.12
AFCS Dual Port RAM	All	4,096	1,184	2,912	28.91

Figure I-21. Capacity Information.

I-22

STATEMENT: The impact of changes to timing can be easily determined.

CHARACTERISTIC: Expandability

EXPLANATION: Since modifications to the system may result in additional consumption of processor time or data bus bandwidth, maintainers must be able to easily determine the timing impacts of changes. Maintenance personnel should have procedures/tools available to inform them of any timing impacts to the system resulting from software changes.

GLOSSARY:
FRAME TIME. A predetermined period of time during which specific functions must be performed.
TIMING SCHEME. The implementation of the timing requirements with consideration to time slicing, time sharing, priority levels, or rate groups as applied to the overall sequencing and execution of program functions.
DATA BUS BANDWIDTH. The capacity of the communications channel used to connect the CPU, memory, and peripherals that is used to carry data. The width of the data bus is one of the main factors determining the processing power of a computer. Most current (circa 1995) processor designs use a 32-bit bus, meaning that 32 bits of data can be transferred at once.

SPECIAL RESPONSE INSTRUCTIONS: Answer N/A if there are clearly no timing implications in this CSCI.

EXAMPLES: Figure I-22 for a flight control computer might be used to determine the timing impacts of changes after the fact. Maintainers should be able to get this information easily and know what the reserve limits are.

FRAME X OF 60 FOR HFS PROCESSOR	MICROSECONDS PROCESSED (OF 66667)	% SPARE (50 REQUIRED)	FAILURE?
1	29,184	56.22	No
2	29,612	55.58	No
3	27,176	59.24	No
4	35,484	49.77	Yes
5	24,200	63.70	No
.	.	.	.
.	.	.	.
.	.	.	.
60	31,560	52.66	No

Figure I-22. Timing and Frame Capacity.

I-23

STATEMENT: Conservation of processor time and/or data bus bandwidth is not a factor in this CSCI.

CHARACTERISTIC: Expandability

EXPLANATION: Complicated timing schemes or restrictions on the use of processor time or data bus bandwidth make the maintainers task more difficult.

GLOSSARY:
TIMING SCHEME. The implementation of the timing requirements with consideration to time slicing, time sharing, priority levels, or rate groups as applied to the overall sequencing and execution of program functions.

SPECIAL RESPONSE INSTRUCTIONS: Answer 6 if there are clearly no timing implications in this CSCI.

EXAMPLE: None

I-24

STATEMENT: Conservation of memory and/or secondary storage is not a factor in this CSCI.

CHARACTERISTIC: Expandability

EXPLANATION: Complicated memory maps or restrictions on the use of memory and/or secondary storage make the maintainer's task more difficult.

GLOSSARY:
MEMORY MAP. A diagram that shows where programs and data are stored in a computer's memory.
SECONDARY STORAGE. Storage facilities forming not an integral part of the computer but directly linked to and controlled by the computer, e.g., disks, magnetic tapes, etc.

SPECIAL RESPONSE INSTRUCTIONS: Answer 6 if there are clearly no sizing implications in this CSCI.

EXAMPLE: None

I-25

STATEMENT: Overall, it appears that this CSCI's implementation contributes to the maintainability of the system.

CHARACTERISTIC: General

EXPLANATION: Consider all aspects of the implementation including any other aspects that were not covered in this questionnaire. In addition to a 6 through 1 answer, provide any comments that reflect the overall maintainability of the CSCI.

EXAMPLE: None

GLOSSARY: None

SPECIAL RESPONSE INSTRUCTIONS: None

GLOSSARY OF TERMS

Where possible, definitions for terms were derived from the following sources: IEEE Standard Glossary of Software Engineering Terminology (IEEE STD 610.12-1990), MIL-STD-498, and DoD-STD-2167A.

ACTUAL PARAMETER. An argument or expression used within a call to a subprogram to specify data or program elements to be transmitted to the subprogram. Contrast with formal parameter.

ADDRESS. (1) A character or group of characters that identifies a register, a particular part of storage, or some other data source or destination. (2) To refer to a device or an item of data.

ALGORITHM. (1) A finite set of well-defined rules for the solution of a problem in a finite number of steps; for example, a complete specification of the sequence of arithmetic operations for evaluating sin x to a given precision. (2) Any sequence of operations for performing a specific task.

ARCHITECTURAL DESIGN. The process of defining a collection of hardware and software components and their interfaces to establish a framework for the development of a computer system.

ARRAY. An n-dimensional ordered set of data items identified by a single name and one or more indices, so that each element of the set is individually addressable. For example, a matrix, table or vector.

ASSEMBLY LANGUAGE. A computer-oriented language whose instructions are usually one-to-one correspondence with computer instructions and that may provide facilities such as the use of macro instructions.

BASIC ALGEBRA. Functions (including trigonometric and geometric functions), equations, polynomials, graphing of functions, basic manipulations, Booleans, etc. This specifically excludes calculus, differential equations, Fourier transforms, statistical techniques, and control theory.

BRANCHING. Nonsequential execution based on control statements (e.g., IF-THEN, looping, or CASE) or explicit transfers of control (e.g., GOTO).

COHESION. The manner and degree to which the tasks performed by a single software component are related to one another. Types include coincidental, communicational, functional, logical, procedural, sequential, and temporal.

COMPILE. To translate a higher order language program to its relocatable or absolute machine code equivalent.

COMPLEX EXPRESSION. A statement or declaration that expresses a formula for calculating a value. A complex expression is considered difficult, and composed of expressions and operators.

COMPLEX MATHEMATICAL MODEL. Examples of complex mathematical models are Fourier transform, Laplace transform, numerical integration/differentiation, control theory, statistical techniques.

COMPOSITE DATA STRUCTURE. An instantiation of a composite data type.

COMPOSITE DATA TYPE. A data type, each of whose members are composed of multiple data items. For example, a data type called PAIRS whose members are ordered pairs (x,y).

COMPUTER SOFTWARE CONFIGURATION ITEM (CSCI). An aggregation of software that satisfies an end-use function is designated for configuration management and is treated as a single entity in the configuration management process.

COMPUTER SOFTWARE UNIT (CSU). An element specified in the design of a CSCI that is separately testable and contains a group of modules that are (usually) functionally related. As a rule of thumb, a CSU is the level at which a SDF is maintained.

CONCURRENT PROCESSES. Processes that may execute in parallel on multiple processors or asynchronously on a single processor. Concurrent processes may interact with each other, and one process may suspend execution pending receipt of information from another process or the occurrences of an external event.

CONTROL FLOW. The sequence in which operations are performed during the execution of a computer program.

CONTROL FLOW DIAGRAM. A diagram that depicts the set of all possible sequences in which operations may be performed during the execution of a system or program. Types include box diagram, flowchart, input-process-output chart, state diagram.

CONTROL STATEMENT. A program statement that selects among alternative sets of program statements or affects the order in which operations are performed. For example, IF-THEN-ELSE, CASE.

CONVENTIONS. Requirements employed to prescribe a disciplined uniform approach to providing consistency in a software product, that is, uniform patterns or forms for arranging data.

COUPLING. The manner and degree of interdependence between software components. Types include common-environmental coupling, data coupling, hybrid coupling, and pathological coupling.

DATA ABSTRACTION. The process of extracting the essential characteristics of data by defining data types and their associated functional characteristics, and disregarding representation details.

DATA BUS BANDWIDTH. The capacity of the communications channel used to connect the CPU, memory, and peripherals that is used to carry data. The width of the data bus is one of the main factors determining the processing power of a computer. Most current (circa 1995) processor designs use a 32-bit bus, meaning that 32 bits of data can be transferred at once.

DATA DICTIONARY. A collection of the names of all data items used in a software system, together with relevant attributes of those items.

DATA FLOW. The sequence in which data transfer, use and transformation are performed during the execution of a computer program.

DATA FLOW DIAGRAM. A diagram that depicts data sources, data sinks, data storage, and processes performed on data as nodes, and logical flow of data as links between the nodes.

DATA ITEM. Any parameter, variable, or constant.

DATA STRUCTURE. A physical or logical relationship among data elements, designed to support specific data manipulation functions.

DATABASE. A collection of interrelated data stored together in one or more computerized files.

DYNAMIC ALLOCATION. A computer resource allocation technique in which the resources assigned to a program vary during program execution, based on current need.

EMBEDDED COMMENTS. Comments exclusive of those in a preface block.

EMBEDDED SYSTEM. A computer system that is part of a larger system and performs some of the requirements of that system; for example, a computer system used in an aircraft or rapid transit system.

ENCAPSULATION. A software development technique that consists of isolating a system function or a set of data and operations on those data within a module and providing precise specifications for the module.

ERROR PROCESSING. The steps required to set program data and control states following the detection of an undesirable event.

ESOTERIC. Designed for or understood by few.

EXCEPTION. An event that causes suspension of normal program execution.

EXECUTABLE STATEMENT. A statement as defined by a high-order language excluding comments, data declarations, and variable/constant declarations.

EXTERNAL INTERFACE. A boundary at which independent computer programs interact or at which a computer program interacts with hardware (e.g., program input/output data or interrupts) or users (e.g., human-computer interface).

EXTRANEOUS CODE. Code not used or that can never be executed, variables never referenced, types not used, or included software components never used.

FAN-IN. A measure of the number of software components that directly control a given software component.

FAN-OUT. A measure of the number of software components that are directly controlled by a given software component.

FORMAL PARAMETER. A variable used in a software module to represent data or program elements that are to be passed to the module by a calling module.

FOURTH GENERATION LANGUAGE (4GL). A computer language designed to improve the productivity achieved by high-order languages and, often, to make computing power available to nonprogrammers. Features typically include an integrated database management system, query language, report generator, and screen definition facility. Additional features may include a graphics generator, decision support capability, and statistical analysis functions.

FRAME TIME. A predetermined period of time during which specific functions must be performed.

GLOBAL DATA ITEM. Data that can be accessed by two or more nonnested modules of a computer program without being explicitly passed as parameters between the modules.

GROWTH MEMORY. Ability to add more memory without changing the architecture of the system.

HIGH-ORDER LANGUAGE (HOL). A programming language that requires little knowledge of the computer on which it will run, can be translated into several different machine languages, allows symbolic naming, and usually results in several machine instructions for each program statement.

INFORMATION HIDING. A software development technique in which each module's interfaces reveal as little as possible about the module's inner workings and other modules are prevented from using information about the module that is not in the module's interface specification.

INITIALIZATION. The preparatory steps required to set the initial program data and control states.

INPUTS. (1) Pertaining to data received from an external source. (2) Pertaining to a device, process, or channel involved in receiving data from an external source. (3) To receive data from an external source. (includes values returned from function calls, parameters, or any global data) (4) To provide data from an external source.

INTEGRATION TEST PROCEDURES. Set of test procedures designed to test a group of functionally related units.

INTERFACE. A shared boundary between the software and other software, hardware, or users.

INTERRUPTS. A suspension of a computer program caused by an event external to that program, and performed in such a way that the program can be resumed.

LABEL. A name or identifier assigned to a computer program statement to enable other statements to refer to that statement.

LANGUAGE EXTENSION. Feature in a programming language that is not in the language's standard (i.e., ANSI Standard, ISO Standard.) For example, Microsoft C++ has the statements CIN and COUT that are not part of C++'s ANSI Standard.

LOCAL DATA. Data that can be accessed by only one module or set of nested modules in a computer program.

LOW-LEVEL LANGUAGE. A programming language that corresponds closely to the instruction set of a given computer.

MACHINE DEPENDENCIES. Pertaining to software that relies on features unique to a particular type of computer and, therefore, executes only on computers of that type.

MCCABE'S CYCLOMATIC COMPLEXITY. A measure of the number of decision paths in a software module. The program control flow is represented as a graph, with each node representing a basic block of code (containing no decisions) and each edge (connecting line) representing a control transfer. The cyclomatic complexity is defined as the number of edges minus the number of nodes plus 2 (e - n + 2). In practice, one can determine the cyclomatic complexity by simply counting the number of decisions.

MEMORY MAP. A diagram that shows where programs and data are stored in a computer's memory.

MODULE. The smallest callable component part of a software product. A module is a logically separable part of the program.

NESTING. Incorporating a control statement or control statements into another control statement.

OPERAND. A variable, constant, or function upon which an operation is to be performed. For example, in the expression A = B + 3, A, B and 3 are the operands.

OPERATOR. A mathematical or logical symbol that represents an action to be performed in an operation. For example, in the expression A = B + 3, + is the operator representing addition and the = is the operator representing assignment.

OUTPUTS. (1) Pertaining to data transmitted to an external destination. (2) Pertaining to a device, process, or channel involved in transmitting data to an external destination. (3) To transmit data to an external destination.

OVERLAY. A storage allocation technique in which computer program segments are loaded from auxiliary storage to main storage when needed, overwriting other segments not currently in use.

PARAMETER. (1) A variable that is given a constant value for a specified application. (2) A constant, variable, or expression that is used to pass values between software routines. See actual parameters and formal parameters.

PARAMETERIZED. Referenced by name, not by an actual value (e.g., PI instead of 3.1415926).

PREFACE BLOCK. Separate set of contiguous comments that usually precede the main body of the software component (a similar block at any other common software component location would be satisfactory) and that provide identification information, processing information, and interface information for the software component.

PRODUCTION REPRESENTATIVE. Source code or documentation that is not expected to undergo major changes that alter its maintainability characteristics prior to operational use.

PROGRAM SUPPORT TOOLS. A collection of tools providing automated support for the software system. This may include: General debug aids, test/retest software, trace software/hardware features, use of compiler, link editor,

library management, configuration management, text editor, or text display software tools. Sometimes referred to as Computer Aided Software Engineering (CASE), Software Engineering Environment (SEE), etc.

REAL-TIME SYSTEM. Software that measures, analyzes, or controls real-world events as they occur. A real-time system must respond within strict time constraints.

RECOVERY. The restoration of a system, program, database, or other system resource to a state in which it can perform required functions.

RECURSIVE PROGRAMMING. A process in which a software module calls itself.

REENTRANT PROGRAMMING. Pertaining to a software module that can be entered as part of one process while also in execution as part of another process and still achieve the desired results.

RESOURCE ALLOCATION. The assignment of computer resources to current and waiting jobs; for example, the assignment of main memory, input/output devices, and auxiliary storage to jobs executing concurrently in a computer system.

SCALAR DATA TYPE. A data type that represents a single element of information.

SECONDARY STORAGE. Storage facilities forming not an integral part of the computer but directly linked to and controlled by the computer, e.g., disks, magnetic tapes, etc.

SHARED MEMORY. Memory locations that can be referenced by two or more different variable names.

SIZING. The process of estimating the amount of computer storage or the number of source lines required for a software system or software component.

SOFTWARE COMPONENT. A module or CSU.

SOFTWARE DEVELOPMENT FILE (SDF). A repository for material pertinent to the development of a particular body of software. Contents typically include (either directly or by reference) considerations, rationale, and constraints related to requirements analysis, design, and implementation; developer-internal test information; and scheduled and status information.

SPARE MEMORY. Installed memory that remains unused (but available) under peak loading conditions.

SPECIALIZED CODING. Code that is machine dependent (e.g., masking, bit manipulation, embedded assembly code).

STANDARDS. Mandatory requirements employed and enforced to prescribe a disciplined uniform approach to software development, that is, mandatory conventions and practices are, in fact, standards.

STRUCTURED DESIGN. Any disciplined approach to software design that adheres to specified rules based on principles such as modularity, top-down design, and stepwise refinement of data, system structures, and processing steps.

STRUCTURED PROGRAM. A computer program constructed of a basic set of control structures, each having one entry and one exit. The set of control structures typically includes: a sequence of two or more instructions, conditional selection of one of two or more instructions, and repetition of a sequence of instructions.

STRUCTURED PROGRAMMING. Any software development technique that includes structured design and results in the development of structured programs.

TERMINATION. The steps required to set the final program data and control states (due to normal or abnormal termination).

TEST PLAN. A document prescribing the approach to be taken for intended testing activities. The plan typically identifies the items to be tested, the testing to be performed, test schedules, personnel requirements, reporting requirements, evaluation criteria, and any risks requiring contingency plans.

TEST PROCEDURE. Detailed instructions for the setup operation, and evaluation of results for a given test. A set of test procedures is often combined to form a test procedures document.

TIMING SCHEME. The implementation of the timing requirements with consideration to time slicing, time sharing, priority levels, or rate groups as applied to the overall sequencing and execution of program functions.

UNIT TEST PROCEDURES. Set of test procedures designed to test an individual unit.

ASETS EVALUATION REQUEST FORM

Program: _____ CSCI Name*: _____
DSE: _____ Language and Dialect: _____
STM/SAS POC: _____ Development Platform: _____
Date Submitted: _____ Compiler and Extensions: _____
Sample Needed By**: _____ Total # of Modules Expected: _____
Modules Needed for Evaluation (Sample Size): _____

Media Information (Circle one)***:

a) 8 mm UNIX tape
b) 4 mm UNIX tape
c) 1/4" UNIX tape
d) 3 1/2" diskette
e) VAX-TK50
f) VAX-TK70
g) 9-track

Access Format (Circle one)***:

a) FTP
b) Sun UNIX tar
c) Sun UNIX cpio
d) Sun UNIX cp
e) Sun UNIX dump
f) VAX-backup

Additional Information:

*All code sent to SAS MUST be compilable at this location. Be sure to include ALL compilation files, makefiles, system-specific packages or libraries, etc. needed to perform a complete compile of the code.
**Please allow 6 weeks for processing.
***These items are ordered based on the ease of use by AFOTEC/SAS. The first item represents the easiest format for us to work with. Formats outside of this list are unacceptable.

SAS USE ONLY

Tools Used: _____
_____
_____
_____

# of Modules Processed: _____
Percent of System: _____

Evaluator Comments:

STANDARD QUESTIONNAIRE ANSWER SHEETS

AFOTEC Pamphlet 99-102, Volume 3
Maintainability Evaluation Scoresheet

Documentation

Evaluator:	System:
Date:	Language:
Location:	CSCI:
DoD/Mil-Std	WINQAS Code:

D-1: 1 2 3 4 5 6 N/A	D-12: 1 2 3 4 5 6 N/A	D-23: 1 2 3 4 5 6 N/A
D-2: 1 2 3 4 5 6 N/A	D-13: 1 2 3 4 5 6 N/A	D-24: 1 2 3 4 5 6 N/A
D-3: 1 2 3 4 5 6 N/A	D-14: 1 2 3 4 5 6 N/A	D-25: 1 2 3 4 5 6 N/A
D-4: 1 2 3 4 5 6 N/A	D-15: 1 2 3 4 5 6 N/A	D-26: 1 2 3 4 5 6 N/A
D-5: 1 2 3 4 5 6 N/A	D-16: 1 2 3 4 5 6 N/A	D-27: 1 2 3 4 5 6 N/A
D-6: 1 2 3 4 5 6 N/A	D-17: 1 2 3 4 5 6 N/A	D-28: 1 2 3 4 5 6 N/A
D-7: 1 2 3 4 5 6 N/A	D-18: 1 2 3 4 5 6 N/A	D-29: 1 2 3 4 5 6 N/A
D-8: 1 2 3 4 5 6 N/A	D-19: 1 2 3 4 5 6 N/A	D-30: 1 2 3 4 5 6 N/A
D-9: 1 2 3 4 5 6 N/A	D-20: 1 2 3 4 5 6 N/A	D-31: 1 2 3 4 5 6 N/A
D-10: 1 2 3 4 5 6 N/A	D-21: 1 2 3 4 5 6 N/A	D-32: 1 2 3 4 5 6 N/A
D-11: 1 2 3 4 5 6 N/A	D-22: 1 2 3 4 5 6 N/A	

Comments

Documentation (Continued)

Comments

Module

Evaluator:	CSCI:
Date:	CSC:
Location:	CSU:
Language:	Module:
System:	WINQAS Code:

M-1: 1 2 3 4 5 6 N/A	M-13: 1 2 3 4 5 6 N/A	M-25: 1 2 3 4 5 6 N/A
M-2: 1 2 3 4 5 6 N/A	M-14: 1 2 3 4 5 6 N/A	M-26: 1 2 3 4 5 6 N/A
M-3: 1 2 3 4 5 6 N/A	M-15: 1 2 3 4 5 6 N/A	M-27: 1 2 3 4 5 6 N/A
M-4: 1 2 3 4 5 6 N/A	M-16: 1 2 3 4 5 6 N/A	M-28: 1 2 3 4 5 6 N/A
M-5: 1 2 3 4 5 6 N/A	M-17: 1 2 3 4 5 6 N/A	M-29: 1 2 3 4 5 6 N/A
M-6: 1 2 3 4 5 6 N/A	M-18: 1 2 3 4 5 6 N/A	M-30: 1 2 3 4 5 6 N/A
M-7: 1 2 3 4 5 6 N/A	M-19: 1 2 3 4 5 6 N/A	M-31: 1 2 3 4 5 6 N/A
M-8: 1 2 3 4 5 6 N/A	M-20: 1 2 3 4 5 6 N/A	M-32: 1 2 3 4 5 6 N/A
M-9: 1 2 3 4 5 6 N/A	M-21: 1 2 3 4 5 6 N/A	M-33: 1 2 3 4 5 6 N/A
M-10: 1 2 3 4 5 6 N/A	M-22: 1 2 3 4 5 6 N/A	M-34: 1 2 3 4 5 6 N/A
M-11: 1 2 3 4 5 6 N/A	M-23: 1 2 3 4 5 6 N/A	M-35: 1 2 3 4 5 6 N/A
M-12: 1 2 3 4 5 6 N/A	M-24: 1 2 3 4 5 6 N/A	

Comments

Module (Continued)

Comments

Computer Software Unit (CSU)	
Evaluator:	System:
Date:	CSCI:
Location:	CSC:
Language:	CSU:
	WINQAS Code:

C-1: 1 2 3 4 5 6 N/A	C-5: 1 2 3 4 5 6 N/A	C-9: 1 2 3 4 5 6 N/A
C-2: 1 2 3 4 5 6 N/A	C-6: 1 2 3 4 5 6 N/A	C-10: 1 2 3 4 5 6 N/A
C-3: 1 2 3 4 5 6 N/A	C-7: 1 2 3 4 5 6 N/A	C-11: 1 2 3 4 5 6 N/A
C-4: 1 2 3 4 5 6 N/A	C-8: 1 2 3 4 5 6 N/A	C-12: 1 2 3 4 5 6 N/A

Comments

Computer Software Unit (Continued)

Comments

Implementation

Evaluator:	System:
Date:	Language:
Location:	CSCI:
DoD/Mil-Std:	WINQAS Code:

I-1:	1 2 3 4 5 6 N/A	I-10:	1 2 3 4 5 6 N/A	I-19:	1 2 3 4 5 6 N/A
I-2:	1 2 3 4 5 6 N/A	I-11:	1 2 3 4 5 6 N/A	I-20:	1 2 3 4 5 6 N/A
I-3:	1 2 3 4 5 6 N/A	I-12:	1 2 3 4 5 6 N/A	I-21:	1 2 3 4 5 6 N/A
I-4:	1 2 3 4 5 6 N/A	I-13:	1 2 3 4 5 6 N/A	I-22:	1 2 3 4 5 6 N/A
I-5:	1 2 3 4 5 6 N/A	I-14:	1 2 3 4 5 6 N/A	I-23:	1 2 3 4 5 6 N/A
I-6:	1 2 3 4 5 6 N/A	I-15:	1 2 3 4 5 6 N/A	I-24:	1 2 3 4 5 6 N/A
I-7:	1 2 3 4 5 6 N/A	I-16:	1 2 3 4 5 6 N/A	I-25:	1 2 3 4 5 6. N/A
I-8:	1 2 3 4 5 6 N/A	I-17:	1 2 3 4 5 6 N/A		
I-9:	1 2 3 4 5 6 N/A	I-18:	1 2 3 4 5 6 N/A		

Comments

AFOTEC Pamphlet 99-102, Volume 3
Maintainability Evaluation Scoresheet

Implementation (Continued)

Comments

DOCUMENT NAME MAP BETWEEN MIL-STD-498 AND OTHER STANDARDS

MIL-STD-498 DID	DoD-STD-2167A AND DOD-STD-7935A SOURCE DIDS
Software Development Plan (SDP)	2167A Software Development Plan (SDP) 7935A Functional Description (FD), section 7
Software Installation Plan (SIP)	7935A Implementation Procedures (IP)
Software Transition Plan (STrP)	2167A Comp Res Integ Sup Doc (CRISD) - planning info 7935A Maintenance Manual (MM) - planning info
Operational Concept Description (OCD)	2167A System/Segment Design Doc (SSDD), section 3 7935A Functional Description (FD), section 2
System/Subsystem Specification (SSS)	2167A System/Segment Specification (SSS) 7935A Functional Description (FD) - system req't info 7935A System/Subsystem Spec (SS) - system req't info
System/Subsystem Design Description (SSDD)	2167A System/Segment Design Document (SSDD) 7935A System/Subsystem Spec - system design info
Software Requirements Specification (SRS)	2167A Software Requirements Specification (SRS) 7935A Software Unit Specification (US) - req't info
Interface Requirements Specification (IRS)	2167A Interface Requirements Specification (IRS) 7935A SW Unit Specification (US) - interface req't info
Software Design Description (SDD)	2167A Software Design Document (SDD) 7935A Software Unit Specification (US) - design info 7935A Maintenance Manual (MM) - "as built" design info
Interface Design Description (IDD)	2167A Interface Design Document (IDD) 7935A SW Unit Specification (US) - interface design info
Database Design Description (DBDD)	7935A Database Specification (DS)
Software Test Plan (STP)	2167A Software Test Plan (STP) 7935A Test Plan (PT) - high-level information
Software Test Description (STD)	2167A Software Test Description (STD) 7935A Test Plan (PT) - detailed information
Software Test Report (STR)	2167A Software Test Report (STR) 7935A Test Analysis Report (RT)
Software Product Specification (SPS)	2167A Software Product Specification (SPS) 2167A CRISD - modification procedures 7935A MM - maintenance procedures
Software Version Description (SVD)	2167A Version Description Document (VDD)
Software User Manual (SUM)	2167A Software User's Manual (SUM) 7935A End User Manual (EM)
Software Center Operator Manual (SCOM)	7935A Computer Operation Manual (OM)
Software Input/Output Manual (SIOM)	7935A Users Manual (UM)
Computer Operation Manual (COM)	2167A Computer System Operator's Manual (CSOM)
Computer Programming Manual (CPM)	2167A Software Programmer's Manual (SPM)
Firmware Support Manual (FSM)	2167A Firmware Support Manual (FSM)

Duplicated from figure 17, MIL-STD-498, 5 December 1994

ACRONYMS

4GL	Fourth Generation Language
AFOTEC	Air Force Operational Test and Evaluation Center
AFOTECP	AFOTEC Pamphlet
ASETS	Automated Software Evaluation Tools System
CASE	Computer Aided Software Engineering
CDR	Critical Design Review
CLRCMP	Computer Resource Life Cycle Management Plan
COI	Critical Operational Issue
CPM	Computer Programming Manual
CRISD	Computer Resource Integrated Support Document
CRISP	Computer Resource Integrated Support Plan
CSC	Computer Software Component
CSCI	Computer Software Configuration Item
CSU	Computer Software Unit
DBDD	Database Design Description
DFD	Data Flow Diagram
DID	Data Item Description
DoD	Department of Defense
DSE	Deputy for Software Evaluation
FQAS	Field Questionnaire Analysis System
HIPO	Hierarchical Input-Processing-Output
HOL	High-Order Language
HQ	Headquarters
HWCI	Hardware Configuration Item
ICD	Interface Control Document
IDD	Interface Design Description
ILS	Integrated Logistics Support
I/O	Input/Output
IRS	Interface Requirements Specification
MOE	Measure of Effectiveness
MOP	Measure of Performance
N/A	Not Applicable
OPR	Office of Primary Responsibility
ORD	Operational Requirements Document
OT&E	Operational Test and Evaluation
PDL	Program Design Language
PDR	Preliminary Design Review
SAS	Software Analysis Team (HQ AFOTEC/SAS)
SDD	Software Design Description
SDF	Software Development File
SDP	Software Development Plan
SIP	Software Installation Plan
SOA	Software Operational Assessment
SPO	System Program Office
SPS	Software Product Specification
SRS	Software Requirements Specification
SSA	Software Support Activity
SSDD	System/Subsystem Design Description

ACRONYMS (continued)

SSS	System/Subsystem Specification
STD	Software Test Description
STM	Software Test Manager
STP	Software Test Plan
STR	Software Test Report
SVD	Software Version Description
TDY	Temporary Duty
TEMP	Test and Evaluation Master Plan
TRP	Test Resources Plan
VDD	Version Description Document
WINQAS	Windows Questionnaire Analysis System